Aprendiste todo sobre recursión en el último capítulo. Este capítulo se enfoca en usar tu nueva habilidad para resolver problemas. Exploraremos divide y vencerás (D&C), una técnica recursiva bien conocida para resolver problemas.

Este capítulo realmente entra en la carne de los algoritmos. Después de todo, un algoritmo no es muy útil si solo puede resolver un tipo de problema. En su lugar, D&C te da una nueva forma de pensar sobre resolver problemas. D&C es otra herramienta en tu caja de herramientas. Cuando obtienes un nuevo problema, no tienes que quedarte perplejo. En su lugar, puedes preguntar, "¿Puedo resolver esto si uso divide y vencerás?"

Al final del capítulo, aprenderás tu primer algoritmo principal de D&C: quicksort. Quicksort es un algoritmo de ordenamiento, y uno mucho más rápido que selection sort (que aprendiste en el capítulo 2). Es un buen ejemplo de código elegante.

**Divide y Vencerás**

D&C puede tomar algo de tiempo para entender. Así que haremos tres ejemplos. Primero te mostraré un ejemplo visual. Luego haré un ejemplo de código que es menos bonito pero tal vez más fácil. Finalmente, repasaremos quicksort, un algoritmo de ordenamiento que usa D&C.

Supón que eres un granjero con una parcela de tierra.

Quieres dividir esta granja uniformemente en parcelas cuadradas. Quieres que las parcelas sean lo más grandes posible. Así que ninguna de estas funcionará.

¿Cómo averiguas el tamaño cuadrado más grande que puedes usar para una parcela de tierra? ¡Usa la estrategia D&C! Los algoritmos D&C son algoritmos recursivos. Para resolver un problema usando D&C, hay dos pasos:

1. Averigua el caso base. Este debería ser el caso más simple posible.
2. Divide o disminuye tu problema hasta que se convierta en el caso base.

Usemos D&C para encontrar la solución a este problema. ¿Cuál es el tamaño cuadrado más grande que puedes usar?

Primero, averigua el caso base. El caso más fácil sería si un lado fuera múltiplo del otro lado.

Supón que un lado es 25 metros (m) y el otro lado es 50 m. Entonces la caja más grande que puedes usar es 25 m × 25 m. Necesitas dos de esas cajas para dividir la tierra.

Ahora necesitas averiguar el caso recursivo. Aquí es donde entra D&C. Según D&C, con cada llamada recursiva, tienes que reducir tu problema. ¿Cómo reduces el problema aquí? Comencemos marcando las cajas más grandes que puedes usar.

Puedes ajustar dos cajas de 640 × 640 ahí, y hay algo de tierra que aún queda por dividir. Ahora viene el momento "¡Ajá!" Hay un segmento de granja que queda por dividir. ¿Por qué no aplicar el mismo algoritmo a este segmento?

Así que comenzaste con una granja de 1680 × 640 que necesitaba ser dividida. Pero ahora necesitas dividir un segmento más pequeño, 640 × 400. Si encuentras la caja más grande que funcionará para este tamaño, esa será la caja más grande que funcionará para toda la granja. ¡Acabas de reducir el problema de una granja de 1680 × 640 a una granja de 640 × 400!

**Algoritmo de Euclides**

"Si encuentras la caja más grande que funcionará para este tamaño, esa será la caja más grande que funcionará para toda la granja." Si no es obvio para ti por qué esta declaración es verdad, no te preocupes. No es obvio. Desafortunadamente, la prueba de por qué funciona es un poco demasiado larga para incluir en este libro, así que tendrás que creerme que funciona. Si quieres entender la prueba, busca el algoritmo de Euclides para encontrar el Máximo Común Divisor. Khan Academy tiene una buena explicación aquí: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm.

**1680 y 640**

Apliquemos el mismo algoritmo de nuevo. Comenzando con una granja de 640 × 400m, la caja más grande que puedes crear es 400 × 400 m. Y eso te deja con un segmento más pequeño, 400 × 240 m.

Y puedes dibujar una caja en eso para obtener un segmento aún más pequeño, 240 × 160 m.

Y luego dibujas una caja en eso para obtener un segmento aún más pequeño.

¡Oye, estás en el caso base: 80 es un factor de 160! Si divides este segmento usando cajas, no te queda nada!

Así que para la granja original, el tamaño de parcela más grande que puedes usar es 80 × 80 m.

Para recapitular, así es como funciona D&C:

1. Averigua un caso simple como el caso base.
2. Averigua cómo reducir tu problema y llegar al caso base.

D&C no es un algoritmo simple que puedas aplicar a un problema. En su lugar, es una forma de pensar sobre un problema. Hagamos un ejemplo más.

Te dan un array de números.

Tienes que sumar todos los números y devolver el total. Es bastante fácil hacer esto con un bucle:

def sum(arr):

total = 0

for x in arr:

total += x

return total

print(sum([1, 2, 3, 4]))

Pero ¿cómo harías esto con una función recursiva?

**Paso 1:** Averigua el caso base. ¿Cuál es el array más simple que podrías obtener? Piensa en el caso más simple, y luego continúa leyendo. Si obtienes un array con 0 o 1 elemento, es bastante fácil sumar.

Así que ese será el caso base.

**Paso 2:** Necesitas acercarte a un array vacío con cada llamada recursiva. ¿Cómo reduces el tamaño de tu problema? Aquí hay una forma.

Es lo mismo que esto.

En cualquier caso, el resultado es 12. Pero en la segunda versión, estás pasando un array más pequeño a la función sum. Es decir, ¡disminuiste el tamaño de tu problema!

Tu función sum podría funcionar así.

Aquí está en acción.

Recuerda, la recursión mantiene seguimiento del estado.

**CONSEJO** Cuando estés escribiendo una función recursiva que involucre un array, el caso base a menudo es un array vacío o un array con un elemento. Si estás atascado, prueba eso primero.

**Vistazo a la programación funcional**

"¿Por qué haría esto recursivamente si puedo hacerlo fácilmente con un bucle?" podrías estar pensando. Bueno, ¡esto es un vistazo a la programación funcional! Los lenguajes de programación funcional como Haskell no tienen bucles, así que tienes que usar recursión para escribir funciones como esta. Si tienes un buen entendimiento de la recursión, los lenguajes funcionales serán más fáciles de aprender. Por ejemplo, aquí está cómo escribirías una función sum en Haskell:

sum [] = 0 #A

sum (x:xs) = x + (sum xs) #B

#A Caso base

#B Caso recursivo

Nota que parece que tienes dos definiciones para la función. La primera definición se ejecuta cuando llegas al caso base. La segunda definición se ejecuta en el caso recursivo. También puedes escribir esta función en Haskell usando una declaración if:

sum arr = if arr == []

then 0

else (head arr) + (sum (tail arr))

Pero la primera definición es más fácil de leer. Como Haskell hace uso intensivo de la recursión, incluye todo tipo de detalles como este para hacer la recursión fácil. Si te gusta la recursión, o estás interesado en aprender un nuevo lenguaje, echa un vistazo a Haskell.

**Ejercicios**

1. **4.1** Escribe el código para la función sum anterior.
2. **4.2** Escribe una función recursiva para contar el número de elementos en una lista.
3. **4.3** Escribe una función recursiva para encontrar el número máximo en una lista.
4. **4.4** ¿Recuerdas la búsqueda binaria del capítulo 1? También es un algoritmo de divide y vencerás. ¿Puedes crear el caso base y el caso recursivo para la búsqueda binaria?

**Quicksort**

Quicksort es un algoritmo de ordenamiento. Es mucho más rápido que selection sort y se usa frecuentemente en la vida real. Quicksort también usa divide y vencerás.

Usemos quicksort para ordenar un array. ¿Cuál es el array más simple que un algoritmo de ordenamiento puede manejar (recuerda mi consejo de la sección anterior)? Bueno, algunos arrays no necesitan ser ordenados en absoluto.

Los arrays vacíos y arrays con solo un elemento serán el caso base. Puedes simplemente devolver esos arrays como están—no hay nada que ordenar:

def quicksort(array):

if len(array) < 2:

return array

Veamos arrays más grandes. Un array con dos elementos es bastante fácil de ordenar también.

¿Qué pasa con un array de tres elementos?

Recuerda, estás usando D&C. Así que quieres desglosar este array hasta que estés en el caso base. Así es como funciona quicksort. Primero, elige un elemento del array. Este elemento se llama el pivote.

Hablaremos sobre cómo elegir un buen pivote más tarde. Por ahora, digamos que el primer elemento en el array es el pivote.

Ahora encuentra los elementos menores que el pivote y los elementos mayores que el pivote.

Esto se llama particionamiento. Ahora tienes:

• Un sub-array de todos los números menores que el pivote • El pivote • Un sub-array de todos los números mayores que el pivote

Los dos sub-arrays no están ordenados. Solo están particionados. Pero si estuvieran ordenados, entonces ordenar todo el array sería bastante fácil.

Si los sub-arrays están ordenados, entonces puedes combinar todo así—array izquierdo + pivote + array derecho—y obtienes un array ordenado. En este caso, es [10, 15] + [33] + [] = [10, 15, 33], que es un array ordenado.

¿Cómo ordenas los sub-arrays? Bueno, el caso base de quicksort ya sabe cómo ordenar arrays vacíos (el sub-array derecho), y puede ordenar recursivamente arrays de dos elementos (el sub-array izquierdo). Así que si llamas quicksort en los dos sub-arrays y luego combinas los resultados, ¡obtienes un array ordenado!

quicksort([15, 10]) + [33] + quicksort([])

> [10, 15, 33] #A

#A Un array ordenado

Esto funcionará con cualquier pivote. Supón que eliges 15 como el pivote en su lugar.

Ambos sub-arrays tienen solo un elemento, y sabes cómo ordenar esos. Así que ahora sabes cómo ordenar un array de tres elementos. Aquí están los pasos:

1. Elige un pivote.
2. Particiona el array en dos sub-arrays: elementos menores que el pivote y elementos mayores que el pivote.
3. Llama quicksort recursivamente en los dos sub-arrays.

¿Qué pasa con un array de cuatro elementos?

Supón que eliges 33 como el pivote de nuevo.

El array de la izquierda tiene tres elementos. Ya sabes cómo ordenar un array de tres elementos: llama quicksort en él recursivamente.

Así que puedes ordenar un array de cuatro elementos. Y si puedes ordenar un array de cuatro elementos, puedes ordenar un array de cinco elementos. ¿Por qué es eso? Supón que tienes este array de cinco elementos.

Aquí están todas las formas en que puedes particionar este array, dependiendo de qué pivote elijas.

Nota que todos estos sub-arrays tienen entre 0 y 4 elementos. ¡Y ya sabes cómo ordenar un array de 0 a 4 elementos usando quicksort! Así que sin importar qué pivote elijas, puedes llamar quicksort recursivamente en los dos sub-arrays.

Por ejemplo, supón que eliges 3 como el pivote. Llamas quicksort en los sub-arrays.

Los sub-arrays se ordenan, y luego combinas todo para obtener un array ordenado. Esto funciona incluso si eliges 5 como el pivote.

Esto funciona con cualquier elemento como el pivote. Así que puedes ordenar un array de cinco elementos. Usando la misma lógica, puedes ordenar un array de seis elementos, y así sucesivamente.

**Pruebas inductivas**

Acabas de obtener un vistazo a las pruebas inductivas! Las pruebas inductivas son una forma de probar que tu algoritmo funciona. Cada prueba inductiva tiene dos pasos: el caso base y el caso inductivo. ¿Suena familiar? Por ejemplo, supón que quiero probar que puedo subir hasta la cima de una escalera. En el caso inductivo, si mis piernas están en un peldaño, puedo poner mis piernas en el siguiente peldaño. Así que si estoy en el peldaño 2, puedo subir al peldaño 3. Ese es el caso inductivo. Para el caso base, diré que mis piernas están en el peldaño 1. Por lo tanto, puedo subir toda la escalera, subiendo un peldaño a la vez.

Usas razonamiento similar para quicksort. En el caso base, mostré que el algoritmo funciona para el caso base: arrays de tamaño 0 y 1. En el caso inductivo, mostré que si quicksort funciona para un array de tamaño 1, funcionará para un array de tamaño 2. Y si funciona para arrays de tamaño 2, funcionará para arrays de tamaño 3, y así sucesivamente. Entonces puedo decir que quicksort funcionará para todos los arrays de cualquier tamaño. No profundizaré más en las pruebas inductivas aquí, pero son divertidas y van de la mano con D&C.

Aquí está el código para quicksort:

def quicksort(array):

if len(array) < 2:

return array #A

else:

pivot = array[0] #B

less = [i for i in array[1:] if i <= pivot] #C

greater = [i for i in array[1:] if i > pivot] #D

return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10, 5, 2, 3]))

#A Caso base: arrays con 0 o 1 elemento ya están "ordenados".

#B Caso recursivo

#C Sub-array de todos los elementos menores que el pivote

#D Sub-array de todos los elementos mayores que el pivote

**Notación Big O revisitada**

Quicksort es único porque su velocidad depende del pivote que elijas. Antes de hablar sobre quicksort, veamos los tiempos de ejecución Big O más comunes de nuevo.

**Estimaciones basadas en una computadora lenta que realiza 10 operaciones por segundo**

Los tiempos de ejemplo en este gráfico son estimaciones si realizas 10 operaciones por segundo. Estos gráficos no son precisos—solo están ahí para darte una sensación de qué tan diferentes son estos tiempos de ejecución. En realidad, tu computadora puede hacer mucho más que 10 operaciones por segundo.

Cada tiempo de ejecución también tiene un algoritmo de ejemplo adjunto. Echa un vistazo a selection sort, que aprendiste en el capítulo 2. Es O(n²). Es un algoritmo bastante lento.

Hay otro algoritmo de ordenamiento llamado merge sort, que es O(n log n). ¡Mucho más rápido! Quicksort es un caso complicado. En el peor caso, quicksort toma tiempo O(n²).

¡Es tan lento como selection sort! Pero ese es el peor caso. En el caso promedio, quicksort toma tiempo O(n log n). Así que podrías estar preguntándote:

¿Qué significan peor caso y caso promedio aquí?

Si quicksort es O(n log n) en promedio, pero merge sort es O(n log n) siempre, ¿por qué no usar merge sort? ¿No es más rápido?

**Merge sort vs. quicksort**

Supón que tienes esta función simple para imprimir cada elemento en una lista:

def print\_items(myList):

for item in myList:

print(item)

Esta función recorre cada elemento en la lista y lo imprime. Como hace bucle sobre toda la lista una vez, esta función se ejecuta en tiempo O(n). Ahora, supón que cambias esta función para que duerma por 1 segundo antes de imprimir un elemento:

from time import sleep

def print\_items2(myList):

for item in myList:

sleep(1)

print(item)

Antes de imprimir un elemento, hará una pausa por 1 segundo. Supón que imprimes una lista de cinco elementos usando ambas funciones.

Ambas funciones hacen bucle a través de la lista una vez, así que ambas son tiempo O(n). ¿Cuál crees que será más rápida en la práctica? Creo que print\_items será mucho más rápida porque no hace pausa por 1 segundo antes de imprimir un elemento. Así que aunque ambas funciones tienen la misma velocidad en notación Big O, print\_items es más rápida en la práctica. Cuando escribes notación Big O como O(n), realmente significa esto.

c es alguna cantidad fija de tiempo que tu algoritmo toma. Se llama la constante. Por ejemplo, podría ser 10 milisegundos \* n para print\_items versus 1 segundo \* n para print\_items2.

Usualmente ignoras esa constante, porque si dos algoritmos tienen diferentes tiempos Big O, la constante no importa. Toma búsqueda binaria y búsqueda simple, por ejemplo. Supón que ambos algoritmos tuvieran estas constantes.

Podrías decir, "¡Wow! La búsqueda simple tiene una constante de 10 milisegundos, pero la búsqueda binaria tiene una constante de 1 segundo. ¡La búsqueda simple es mucho más rápida!" Ahora supón que estás buscando una lista de 4 mil millones de elementos. Aquí están los tiempos.

Como puedes ver, la búsqueda binaria sigue siendo mucho más rápida. Esa constante no hizo diferencia en absoluto.

Pero a veces la constante puede hacer una diferencia. Quicksort versus merge sort es un ejemplo. A menudo, la forma en que se implementan Quicksort y merge sort, si ambos son tiempo O(n log n), quicksort es más rápido. Y quicksort es más rápido en la práctica porque alcanza el caso promedio mucho más a menudo que el peor caso.

Así que ahora te estás preguntando: ¿cuál es el caso promedio versus el peor caso?

**Caso promedio vs. peor caso**

El rendimiento de quicksort depende mucho del pivote que elijas. Supón que siempre eliges el primer elemento como el pivote. Y llamas quicksort con un array que ya está ordenado. Quicksort no verifica si el array de entrada ya está ordenado. Así que seguirá tratando de ordenarlo.

Nota cómo no estás dividiendo el array en dos mitades. En su lugar, uno de los sub-arrays siempre está vacío. Así que la pila de llamadas es realmente larga. Ahora en su lugar, supón que siempre elegiste el elemento del medio como el pivote. Mira la pila de llamadas ahora.

¡Es tan corta! Como divides el array por la mitad cada vez, no necesitas hacer tantas llamadas recursivas. Llegas al caso base más pronto, y la pila de llamadas es mucho más corta.

El primer ejemplo que viste es el escenario de peor caso, y el segundo ejemplo es el escenario de mejor caso. En el peor caso, el tamaño de la pila es O(n). En el mejor caso, el tamaño de la pila es O(log n). Puedes obtener el mejor caso consistentemente, siempre que elijas un elemento aleatorio como el pivote. Continúa leyendo para saber por qué.

Ahora mira el primer nivel en la pila. Eliges un elemento como el pivote, y el resto de los elementos se dividen en sub-arrays. Tocas todos los ocho elementos en el array. Así que esta primera operación toma tiempo O(n). Tocaste todos los ocho elementos en este nivel de la pila de llamadas. Pero en realidad, tocas elementos O(n) en cada nivel de la pila de llamadas.

Incluso si particionas el array de manera diferente, sigues tocando elementos O(n) cada vez.

En este ejemplo, hay niveles O(log n) (la forma técnica de decir eso es, "La altura de la pila de llamadas es O(log n)"). Y cada nivel toma tiempo O(n). Todo el algoritmo tomará tiempo O(n) \* O(log n) = O(n log n). Este es el escenario de mejor caso.

En el peor caso, hay niveles O(n), así que el algoritmo tomará tiempo O(n) \* O(n) = O(n²).

Bueno, ¡adivina qué! Estoy aquí para decirte que el mejor caso también es el caso promedio. Si siempre eliges un elemento aleatorio en el array como el pivote, quicksort se completará en tiempo O(n log n) en promedio. Quicksort es uno de los algoritmos de ordenamiento más rápidos que existen, y es un muy buen ejemplo de D&C.

**Ejercicios**

¿Cuánto tiempo tomaría cada una de estas operaciones en notación Big O?

1. **4.5** Imprimir el valor de cada elemento en un array.
2. **4.6** Duplicar el valor de cada elemento en un array.
3. **4.7** Duplicar el valor de solo el primer elemento en un array.
4. **4.8** Crear una tabla de multiplicación con todos los elementos en el array. Así que si tu array es [2, 3, 7, 8, 10], primero multiplicas cada elemento por 2, luego multiplicas cada elemento por 3, luego por 7, y así sucesivamente.

**Resumen**

D&C funciona desglosando un problema en piezas cada vez más pequeñas. Si estás usando D&C en una lista, el caso base es probablemente un array vacío o un array con un elemento.

Si estás implementando quicksort, elige un elemento aleatorio como el pivote. El tiempo de ejecución promedio de quicksort es O(n log n)!

La constante en una función puede importar a veces. Dados dos algoritmos con el mismo tiempo de ejecución big-O, uno puede ser consistentemente más rápido que el otro. Por eso quicksort es más rápido que merge sort.

La constante casi nunca importa para búsqueda simple versus búsqueda binaria, porque O(log n) es mucho más rápido que O(n) cuando tu lista se hace grande.

**Capítulo 5: Tablas Hash**

**En este capítulo**

• Aprendes sobre tablas hash, una de las estructuras de datos más útiles. Las tablas hash tienen muchos usos; este capítulo cubre los casos de uso comunes. • Aprendes sobre los internos de las tablas hash: implementación, colisiones y funciones hash. Esto te ayudará a entender cómo analizar el rendimiento de una tabla hash.

Supón que trabajas en una tienda de abarrotes. Cuando un cliente compra productos, tienes que buscar el precio en un libro. Si el libro no está alfabetizado, puede tomarte mucho tiempo mirar a través de cada línea individual para manzana. Estarías haciendo búsqueda simple del capítulo 1, donde tienes que mirar cada línea. ¿Recuerdas cuánto tiempo tomaría eso? Tiempo O(n). Si el libro está alfabetizado, podrías ejecutar búsqueda binaria para encontrar el precio de una manzana. Eso solo tomaría tiempo O(log n).

Como recordatorio, hay una gran diferencia entre tiempo O(n) y O(log n)! Supón que pudieras mirar a través de 10 líneas del libro por segundo. Aquí está cuánto tiempo tomarían búsqueda simple y búsqueda binaria.

Ya sabes que la búsqueda binaria es muy rápida. Pero como cajero, buscar cosas en un libro es una molestia, incluso si el libro está ordenado. Puedes sentir al cliente calentándose mientras buscas elementos en el libro. Lo que realmente necesitas es un amigo que tenga todos los nombres y precios memorizados. Entonces no necesitas buscar nada: le preguntas, y te dice la respuesta instantáneamente.

Tu amiga Maggie puede darte el precio en tiempo O(1) para cualquier elemento, sin importar qué tan grande sea el libro. Incluso es más rápida que la búsqueda binaria.

¡Qué persona tan maravillosa! ¿Cómo consigues una "Maggie"? Pongamos nuestros sombreros de estructura de datos. Conoces dos estructuras de datos hasta ahora: arrays y listas (no hablaré de pilas porque realmente no puedes "buscar" algo en una pila). Podrías implementar este libro como un array.

Cada elemento en el array son realmente dos elementos: uno es el nombre de un tipo de producto, y el otro es el precio. Si ordenas este array por nombre, puedes ejecutar búsqueda binaria en él para encontrar el precio de un elemento. Así que puedes encontrar elementos en tiempo O(log n). Pero quieres encontrar elementos en tiempo O(1). Es decir, quieres hacer una "Maggie." Ahí es donde entran las funciones hash.

**Funciones hash**

Una función hash es una función donde pones una cadena¹ y obtienes de vuelta un número.

En terminología técnica, diríamos que una función hash "mapea cadenas a números." Podrías pensar que no hay patrón discernible para qué número obtienes cuando pones una cadena. Pero hay algunos requisitos para una función hash:

• Necesita ser consistente. Por ejemplo, supón que pones "manzana" y obtienes "4". Cada vez que pongas "manzana", deberías obtener "4" de vuelta. Sin esto, tu tabla hash no funcionará.

• Debería mapear palabras diferentes a números diferentes. Por ejemplo, una función hash no es buena si siempre devuelve "1" para cualquier palabra que pongas. En el mejor caso, cada palabra diferente debería mapear a un número diferente.

¹Cadena aquí significa cualquier tipo de datos—una secuencia de bytes.

Así que una función hash mapea cadenas a números. ¿Para qué es bueno eso? Bueno, ¡puedes usarla para hacer tu "Maggie"!

Comienza con un array vacío:

Almacenarás todos tus precios en este array. Agreguemos el precio de una manzana. Alimenta "manzana" en la función hash.

La función hash genera "3". Así que almacenemos el precio de la manzana en el índice 3 en el array.

Agreguemos leche. Alimenta "leche" en la función hash.

La función hash dice "0". Almacenemos el precio de la leche en el índice 0.

Sigue así, y eventualmente todo el array estará lleno de precios.

Ahora preguntas, "Oye, ¿cuál es el precio de un aguacate?" No necesitas buscarlo en el array. Solo alimenta "aguacate" en la función hash.

Te dice que el precio está almacenado en el índice 4. Y efectivamente, ahí está.

La función hash te dice exactamente dónde está almacenado el precio, ¡así que no tienes que buscar en absoluto! Esto funciona porque:

• La función hash mapea consistentemente un nombre al mismo índice. Cada vez que pongas "aguacate", obtendrás el mismo número de vuelta. Así que puedes usarla la primera vez para encontrar dónde almacenar el precio del aguacate, y luego puedes usarla para encontrar dónde almacenaste ese precio.

• La función hash mapea cadenas diferentes a índices diferentes. "Aguacate" mapea al índice 4. "Leche" mapea al índice 0. Todo mapea a un slot diferente en el array donde puedes almacenar su precio.

• La función hash sabe qué tan grande es tu array y solo devuelve índices válidos. Así que si tu array tiene 5 elementos, la función hash no devuelve 100... ese no sería un índice válido en el array.

¡Acabas de construir una "Maggie"! Pon una función hash y un array juntos, y obtienes una estructura de datos llamada tabla hash. Una tabla hash es la primera estructura de datos que aprenderás que tiene algo de lógica extra detrás de ella. Los arrays y listas mapean directamente a memoria, pero las tablas hash son más inteligentes. Usan una función hash para averiguar inteligentemente dónde almacenar elementos.

Las tablas hash son probablemente la estructura de datos compleja más útil que aprenderás. También se conocen como mapas hash, mapas, diccionarios y arrays asociativos. ¡Y las tablas hash son rápidas! Recuerda nuestra discusión de arrays y listas enlazadas en el capítulo 2? Puedes obtener un elemento de un array instantáneamente. Y las tablas hash usan un array para almacenar los datos, así que son igualmente rápidas.

¿Cuál es el truco?

La función hash que acabamos de ver es lo que los muchachos llaman una función hash perfecta. Mapea hábilmente cada elemento de abarrotes a su propio slot en el array:

Míralos todos cómodos en sus propios slots. En realidad, probablemente no obtendrás un mapeo perfecto 1-a-1 como este. Tus elementos necesitarán compartir una habitación. Algunos elementos de abarrotes mapearán al mismo slot, mientras que otros slots quedarán vacíos:

Hay una sección sobre colisiones próximamente que discute esto. Por ahora, solo sabe que mientras las tablas hash son muy útiles, rara vez mapean elementos a slots tan perfectamente.

Por cierto, este tipo de mapeo 1-a-1 se llama una función inyectiva. ¡Úsalo para impresionar a tus amigos!

Probablemente nunca tendrás que implementar tablas hash tú mismo. Cualquier buen lenguaje tendrá una implementación para tablas hash. Python tiene tablas hash; se llaman diccionarios. Puedes hacer una nueva tabla hash usando:

>>> book = {}

book es una nueva tabla hash. Agreguemos algunos precios a book:

>>> book["apple"] = 0.67 #A

>>> book["milk"] = 1.49 #B

>>> book["avocado"] = 1.49

>>> print(book)

{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}

#A Una manzana cuesta 67 centavos.

#B La leche cuesta $1.49.

¡Bastante fácil! Ahora preguntemos por el precio de un aguacate:

>>> print(book["avocado"])

1.49 #A

#A El precio de un aguacate

Una tabla hash tiene claves y valores. En el hash book, los nombres de los productos son las claves, y sus precios son los valores. Una tabla hash mapea claves a valores.

En la siguiente sección, verás algunos ejemplos donde las tablas hash son realmente útiles.

**Ejercicios**

Es importante que las funciones hash devuelvan consistentemente la misma salida para la misma entrada. Si no lo hacen, ¡no podrás encontrar tu elemento después de ponerlo en la tabla hash!

¿Cuáles de estas funciones hash son consistentes?

1. **5.1** f(x) = 1 #A #A Devuelve "1" para toda entrada
2. **5.2** f(x) = random.random() #B #B Devuelve un número aleatorio cada vez
3. **5.3** f(x) = next\_empty\_slot() #C #C Devuelve el índice del siguiente slot vacío en la tabla hash
4. **5.4** f(x) = len(x) #D #D Usa la longitud de la cadena como el índice

**Casos de uso**

Las tablas hash se usan en todas partes. Esta sección te mostrará algunos casos de uso.

**Usando tablas hash para búsquedas**

Tu teléfono tiene una agenda telefónica práctica incorporada.

Cada nombre tiene un número de teléfono asociado con él.

Supón que quieres construir una agenda telefónica como esta. Estás mapeando nombres de personas a números de teléfono. Tu agenda telefónica necesita tener esta funcionalidad:

• Agregar el nombre de una persona y el número de teléfono asociado con esa persona. • Ingresar el nombre de una persona, y obtener el número de teléfono asociado con ese nombre.

¡Este es un caso de uso perfecto para tablas hash! Las tablas hash son geniales cuando quieres:

• Crear un mapeo de una cosa a otra cosa • Buscar algo

Construir una agenda telefónica es bastante fácil. Primero, haz una nueva tabla hash:

>>> phone\_book = {}

Agreguemos los números de teléfono de algunas personas en esta agenda telefónica:

>>> phone\_book["jenny"] = "8675309"

>>> phone\_book["emergency"] = "911"

¡Eso es todo! Ahora, supón que quieres encontrar el número de teléfono de Jenny. Solo pasa la clave a la tabla hash:

>>> print(phone\_book["jenny"])

8675309 #A

#A El número de teléfono de Jenny

Imagina si tuvieras que hacer esto usando un array en su lugar. ¿Cómo lo harías? ¿Cuánto tiempo tomaría? Encuentra las respuestas a estas preguntas en la siguiente sección!

Las tablas hash se usan para búsquedas en una escala mucho mayor. Por ejemplo, supón que vas a un sitio web como http://adit.io. Tu computadora tiene que traducir adit.io a una dirección IP.

Para cualquier sitio web al que vayas, la dirección tiene que ser traducida a una dirección IP.

¡Wow, mapear una dirección web a una dirección IP? ¡Suena como un caso de uso perfecto para tablas hash! Este proceso se llama resolución DNS. Las tablas hash son una forma de proporcionar esta funcionalidad. Tu computadora tiene un caché DNS que almacena este mapeo para sitios web que has visitado recientemente, y una buena forma de construir un caché DNS es usar una tabla hash.

**Prevenir entradas duplicadas**

Supón que estás dirigiendo una casilla de votación. Naturalmente, cada persona puede votar solo una vez. ¿Cómo te aseguras de que no hayan votado antes? Cuando alguien viene a votar, preguntas por su nombre completo. Luego lo verificas contra la lista de personas que han votado.

Si su nombre está en la lista, esta persona ya ha votado—¡échalos! De lo contrario, agregas su nombre a la lista y los dejas votar. Ahora supón que mucha gente ha venido a votar, y la lista de personas que han votado es realmente larga.

Cada vez que alguien nuevo viene a votar, tienes que escanear esta lista gigante para ver si ya han votado. Pero hay una mejor manera: ¡usa una tabla hash!

Primero, haz una tabla hash para mantener seguimiento de las personas que han votado:

>>> voted = {}

Cuando alguien nuevo viene a votar, verifica si ya están en la tabla hash:

>>> value = voted.get("tom")

La función get devuelve el valor si "tom" está en la tabla hash. De lo contrario, devuelve None. ¡Puedes usar esto para verificar si alguien ya ha votado!

Aquí está el código:

voted = {}

def check\_voter(name):

if name in voted:

print("kick them out!")

else:

voted[name] = True

print("let them vote!")

Probémoslo unas cuantas veces:

>>> check\_voter("tom")

let them vote!

>>> check\_voter("mike")

let them vote!

>>> check\_voter("mike")

kick them out!

La primera vez que Tom entra, esto imprimirá, "let them vote!" Luego Mike entra, e imprime, "let them vote!" Luego Mike trata de entrar una segunda vez, e imprime, "kick them out!"

Recuerda, si estuvieras almacenando estos nombres en una lista de personas que han votado, esta función eventualmente se volvería realmente lenta, porque tendría que ejecutar una búsqueda simple sobre toda la lista. Pero estás almacenando sus nombres en una tabla hash en su lugar, y una tabla hash te dice instantáneamente si el nombre de esta persona está en la tabla hash o no.

Verificar duplicados es muy rápido con una tabla hash.

**Usando tablas hash como caché**

Un último caso de uso: caché. Si trabajas en un sitio web, puedes haber oído del caché antes como algo bueno que hacer. Aquí está la idea. Supón que visitas facebook.com:

1. Haces una solicitud al servidor de Facebook.
2. El servidor piensa por un segundo y se le ocurre la página web para enviarte.
3. Obtienes una página web.

Por ejemplo, en Facebook, el servidor puede estar recolectando toda la actividad de tus amigos para mostrarte. Toma un par de segundos recolectar toda esa actividad y mostrártela. Ese par de segundos puede sentirse como mucho tiempo como usuario. Podrías pensar, "¿Por qué Facebook está siendo tan lento?" Por otro lado, los servidores de Facebook tienen que servir a millones de personas, y ese par de segundos se suma para ellos. Los servidores de Facebook están realmente trabajando duro para servir todos esos sitios web. ¿Hay una forma de hacer que los servidores de Facebook hagan menos trabajo?

Supón que tienes una sobrina que sigue preguntándote sobre planetas. "¿Qué tan lejos está Marte de la Tierra?" "¿Qué tan lejos está la Luna?" "¿Qué tan lejos está Júpiter?" Cada vez, tienes que hacer una búsqueda en Google y darle una respuesta. Toma un par de minutos. Ahora, supón que siempre preguntara, "¿Qué tan lejos está la Luna?" Muy pronto, memorizarías que la Luna está a 238,900 millas de distancia. No tendrías que buscarlo en Google... simplemente recordarías y responderías. Así es como funciona el caché: los sitios web recuerdan los datos en lugar de recalcularlos.

Si has iniciado sesión en Facebook, todo el contenido que ves está adaptado solo para ti. Cada vez que vas a facebook.com, sus servidores tienen que pensar sobre qué contenido te interesa. Pero si no has iniciado sesión en Facebook, ves la página de inicio de sesión. Todos ven la misma página de inicio de sesión. Facebook recibe la misma pregunta una y otra vez: "Dame la página de inicio cuando no he iniciado sesión." Así que deja de hacer que el servidor haga trabajo para averiguar cómo se ve la página de inicio. En su lugar, memoriza cómo se ve la página de inicio y te la envía.

Esto se llama caché. Tiene dos ventajas:

Obtienes la página web mucho más rápido, como cuando memorizaste la distancia de la Tierra a la Luna. La próxima vez que tu sobrina pregunte, no tendrás que hacer una búsqueda en Google. Puedes responder instantáneamente.

Facebook tiene que hacer menos trabajo.

El caché es una forma común de hacer las cosas más rápidas. Todos los sitios web grandes usan caché. ¡Y esos datos están cacheados en una tabla hash!

Facebook no solo está cacheando la página de inicio. También está cacheando la página Acerca de, la página de Contacto, la página de Términos y Condiciones, y mucho más. Así que necesita un mapeo de URL de página a datos de página.

Cuando visitas una página en Facebook, primero verifica si la página está almacenada en la tabla hash.

Aquí está en pseudocódigo:

cache = {}

def get\_page(url):

if cache.get(url):

return cache[url] #A

else:

data = get\_data\_from\_server(url)

cache[url] = data #B

return data

#A Devuelve datos cacheados

#B Guarda estos datos en tu caché primero

Aquí, haces que el servidor haga trabajo solo si la URL no está en el caché. Antes de devolver los datos, los guardas en el caché. La próxima vez que alguien solicite esta URL, puedes enviar los datos del caché en lugar de hacer que el servidor haga el trabajo.

**Resumen**

Para recapitular, las tablas hash son buenas para:

• Modelar relaciones de una cosa a otra cosa • Filtrar duplicados • Cachear/memorizar datos en lugar de hacer que tu servidor haga trabajo

**Colisiones**

Como dije antes, la mayoría de los lenguajes tienen tablas hash. No necesitas saber cómo escribir la tuya propia. Así que no hablaré demasiado sobre los internos de las tablas hash. Pero aún te importa el rendimiento! Para entender el rendimiento de las tablas hash, primero necesitas entender qué son las colisiones. Las siguientes dos secciones cubren colisiones y rendimiento.

Primero, te he estado diciendo una mentira piadosa. Te dije que una función hash siempre mapea claves diferentes a slots diferentes en el array.

En realidad, es casi imposible escribir una función hash que haga esto. Tomemos un ejemplo simple. Supón que tu array contiene 26 slots.

Y tu función hash es realmente simple: asigna un lugar en el array alfabéticamente.

Tal vez ya puedas ver el problema. Quieres poner el precio de las manzanas en tu tabla hash. Te asignan el primer slot.

Luego quieres poner el precio de los plátanos en la tabla hash. Te asignan el segundo slot.

¡Todo va tan bien! Pero ahora quieres poner el precio de los aguacates en la tabla hash. Te asignan el primer slot de nuevo.

¡Oh no! Las manzanas ya tienen ese slot! ¿Qué hacer? Esto se llama una colisión: dos claves han sido asignadas al mismo slot. Esta es un problema. Si almacenas el precio de los aguacates en ese slot, sobrescribirás el precio de las manzanas. ¡Entonces la próxima vez que alguien pregunte por el precio de las manzanas, obtendrán el precio de los aguacates en su lugar! Las colisiones son malas, y necesitas trabajar alrededor de ellas. Hay muchas formas diferentes de lidiar con colisiones. La más simple es esta: si múltiples claves mapean al mismo slot, comienza una lista enlazada en ese slot.

En este ejemplo, tanto "apple" como "avocado" mapean al mismo slot. Así que comienzas una lista enlazada en ese slot. Si necesitas saber el precio de los plátanos, sigue siendo rápido. Si necesitas saber el precio de las manzanas, es un poco más lento. Tienes que buscar a través de esta lista enlazada para encontrar "apple". Si la lista enlazada es pequeña, no hay problema—tienes que buscar a través de tres o cuatro elementos. Pero supón que trabajas en una tienda de abarrotes donde solo vendes productos que comienzan con la letra A.

¡Hey, espera un minuto! Toda la tabla hash está completamente vacía excepto por un slot. Y ese slot tiene una lista enlazada gigante! Cada elemento individual en esta tabla hash está en la lista enlazada. Eso es tan malo como poner todo en una lista enlazada para empezar. Va a ralentizar tu tabla hash mucho.

Hay dos lecciones aquí:

• Tu función hash es realmente importante. Tu función hash mapeó todas las claves a un solo slot. Idealmente, tu función hash mapearía claves uniformemente por toda la tabla hash.

• Si esas listas enlazadas se hacen largas, ralentiza tu tabla hash mucho. Pero no se harán largas si usas una buena función hash!

Las funciones hash son importantes. Una buena función hash te dará muy pocas colisiones. Entonces, ¿cómo eliges una buena función hash? ¡Eso viene en la siguiente sección!

**Rendimiento**

Comenzaste este capítulo en la tienda de abarrotes. Querías construir algo que te diera los precios para productos instantáneamente. Bueno, las tablas hash son realmente rápidas.

En el caso promedio, las tablas hash toman O(1) para todo. O(1) se llama tiempo constante. No has visto tiempo constante antes. No significa instantáneo. Significa que el tiempo tomado permanecerá igual, sin importar qué tan grande sea la tabla hash. Por ejemplo, sabes que la búsqueda simple toma tiempo lineal.

La búsqueda binaria es más rápida—toma tiempo log:

Buscar algo en una tabla hash toma tiempo constante.

¿Ves cómo es una línea plana? Eso significa que no importa si tu tabla hash tiene 1 elemento o 1 mil millones de elementos—obtener algo de una tabla hash tomará la misma cantidad de tiempo. En realidad, has visto tiempo constante antes. Obtener un elemento de un array toma tiempo constante. No importa qué tan grande sea tu array; toma la misma cantidad de tiempo obtener un elemento. En el caso promedio, las tablas hash son realmente rápidas.

En el peor caso, una tabla hash toma O(n)—tiempo lineal—para todo, lo que es realmente lento. Comparemos las tablas hash con arrays y listas.

**Nota aparte: otras formas de resolver colisiones**

Hay una forma diferente de manejar colisiones, llamada cuckoo hashing. Con cuckoo hashing, manejas la colisión durante la inserción, no durante la búsqueda. Así que en el peor caso, las búsquedas seguirán siendo tiempo constante.

Mira el caso promedio para tablas hash. Las tablas hash son tan rápidas como los arrays en búsquedas (obteniendo un valor en un índice). Y son tan rápidas como las listas enlazadas en inserciones y eliminaciones. ¡Es lo mejor de ambos mundos! Pero en el peor caso, las tablas hash son lentas en todas esas. Así que es importante que no llegues al rendimiento de peor caso con tablas hash. Y para hacer eso, necesitas evitar colisiones. Para evitar colisiones, necesitas:

• Un factor de carga bajo • Una buena función hash

**NOTA** Antes de comenzar esta siguiente sección, sabe que esto no es lectura requerida. Voy a hablar sobre cómo implementar una tabla hash, pero nunca tendrás que hacer eso tú mismo. Cualquier lenguaje de programación que uses tendrá una implementación de tablas hash incorporada. Puedes usar la tabla hash incorporada y asumir que tendrá buen rendimiento. La siguiente sección te da un vistazo bajo el capó.

**Factor de carga**

El factor de carga de una tabla hash es fácil de calcular.

Las tablas hash usan un array para almacenamiento, así que cuentas el número de slots ocupados en un array. Por ejemplo, esta tabla hash tiene un factor de carga de 2/5, o 0.4.

¿Cuál es el factor de carga de esta tabla hash?

Si dijiste 1/3, tienes razón. El factor de carga mide cuántos slots vacíos quedan en tu tabla hash.

Supón que necesitas almacenar el precio de 100 elementos de productos en tu tabla hash, y tu tabla hash tiene 100 slots. En el mejor caso, cada elemento obtendrá su propio slot.

Esta tabla hash tiene un factor de carga de 1. ¿Qué pasa si tu tabla hash solo tiene 50 slots? Entonces tiene un factor de carga de 2.

No hay forma de que cada elemento obtenga su propio slot, ¡porque no hay suficientes slots! Tener un factor de carga mayor que 1 significa que tienes más elementos que slots en tu array. Una vez que el factor de carga comienza a crecer, necesitas agregar más slots a tu tabla hash. Esto se llama redimensionar. Por ejemplo, supón que tienes esta tabla hash que se está llenando bastante.

Necesitas redimensionar esta tabla hash. Primero creas un nuevo array que sea más grande. La regla general es hacer un array que sea el doble del tamaño.

Ahora necesitas re-insertar todos esos elementos en esta nueva tabla hash usando la función hash:

Esta nueva tabla tiene un factor de carga de 3/8. ¡Mucho mejor! Con un factor de carga más bajo, tendrás menos colisiones, y tu tabla funcionará mejor. Una buena regla general es redimensionar cuando tu factor de carga es mayor que 0.7.

Podrías estar pensando, "¡Este negocio de redimensionar toma mucho tiempo!" Y tienes razón. Redimensionar es caro, y no quieres redimensionar muy seguido. Pero promediado, las tablas hash toman O(1) incluso con redimensionamiento.

**Una buena función hash**

Una buena función hash distribuye valores en el array uniformemente.

Una mala función hash agrupa valores juntos y produce muchas colisiones.

¿Qué es una buena función hash? Eso es algo de lo que nunca tendrás que preocuparte—gente con barbas grandes se sienta en cuartos oscuros y se preocupa por eso. Si tienes mucha curiosidad, busca CityHash. Eso es lo que usa la biblioteca Abseil de Google. Abseil es una biblioteca C++ de código abierto basada en código interno de Google. Proporciona todo tipo de funciones C++ de propósito general. Abseil es un bloque de construcción para el código de Google, así que si usa CityHash, puedes estar seguro de que CityHash es bastante bueno. Podrías usar eso como tu función hash.

**Ejercicios**

Es importante que las funciones hash tengan una buena distribución. Deberían mapear elementos lo más ampliamente posible. El peor caso es una función hash que mapea todos los elementos al mismo slot en la tabla hash.

Supón que tienes estas cuatro funciones hash que funcionan con cadenas:

1. a. Devolver "1" para toda entrada.
2. b. Usar la longitud de la cadena como el índice.
3. c. Usar el primer carácter de la cadena como el índice. Así, todas las cadenas que comienzan con a se hash juntas, y así sucesivamente.
4. d. Mapear cada letra a un número primo: a = 2, b = 3, c = 5, d = 7, e = 11, y así sucesivamente. Para una cadena, la función hash es la suma de todos los caracteres módulo el tamaño del hash. Por ejemplo, si tu tamaño de hash es 10, y la cadena es "bag", el índice es 3 + 2 + 17 % 10 = 22 % 10 = 2.

Para cada uno de estos ejemplos, ¿qué funciones hash proporcionarían una buena distribución? Asume un tamaño de tabla hash de 10 slots.

1. **5.5** Una agenda telefónica donde las claves son nombres y los valores son números de teléfono. Los nombres son los siguientes: Esther, Ben, Bob, y Dan.
2. **5.6** Un mapeo de tamaño de batería a poder. Los tamaños son A, AA, AAA, y AAAA.
3. **5.7** Un mapeo de títulos de libros a autores. Los títulos son Maus, Fun Home, y Watchmen.

**Resumen**

Casi nunca tendrás que implementar una tabla hash tú mismo. El lenguaje de programación que uses debería proporcionar una implementación para ti. Puedes usar las tablas hash de Python y asumir que obtendrás el rendimiento de caso promedio: tiempo constante.

Las tablas hash son una estructura de datos poderosa porque son muy rápidas y te permiten modelar datos de una forma diferente. Puede que pronto te encuentres usándolas todo el tiempo:

• Puedes hacer una tabla hash combinando una función hash con un array. • Las colisiones son malas. Necesitas una función hash que minimice colisiones. • Las tablas hash tienen búsqueda, inserción y eliminación realmente rápidas. • Las tablas hash son buenas para modelar relaciones de un elemento a otro elemento. • Una vez que tu factor de carga es mayor que 0.7, es hora de redimensionar tu tabla hash. • Las tablas hash se usan para cachear datos (por ejemplo, con un servidor web). • Las tablas hash son geniales para atrapar duplicados.

**Capítulo 6: Búsqueda en anchura**

**En este capítulo**

• Aprendes cómo modelar una red usando una nueva estructura de datos abstracta: grafos • Aprendes búsqueda en anchura, un algoritmo que puedes ejecutar en grafos para responder preguntas como, "¿Cuál es el camino más corto para ir a X?" • Aprendes sobre grafos dirigidos versus no dirigidos. • Aprendes ordenamiento topológico, un tipo diferente de algoritmo de ordenamiento que expone dependencias entre nodos.

Este capítulo introduce grafos. Primero, hablaré sobre qué son los grafos (no involucran un eje X o Y). Luego te mostraré tu primer algoritmo de grafos. Se llama búsqueda en anchura (BFS).

La búsqueda en anchura te permite encontrar la distancia más corta entre dos cosas. ¡Pero distancia más corta puede significar muchas cosas! Puedes usar búsqueda en anchura para:

• Escribir un corrector ortográfico (menor número de ediciones de tu error ortográfico a una palabra real—por ejemplo, READED -> READER es una edición) • Encontrar el doctor más cercano a ti en tu red • Construir un rastreador de motor de búsqueda

Los algoritmos de grafos están entre los algoritmos más útiles que conozco. Asegúrate de leer los próximos capítulos cuidadosamente—estos son algoritmos que podrás aplicar una y otra vez.

**Introducción a grafos**

Supón que estás en San Francisco, y quieres ir de Twin Peaks al Puente Golden Gate. Quieres llegar ahí en autobús, con el número mínimo de transbordos. Aquí están tus opciones.

¿Cuál es tu algoritmo para encontrar el camino con menos pasos?

Bueno, ¿puedes llegar ahí en un paso? Aquí están todos los lugares a los que puedes llegar en un paso.

El puente no está resaltado; no puedes llegar ahí en un paso. ¿Puedes llegar ahí en dos pasos?

De nuevo, el puente no está ahí, así que no puedes llegar al puente en dos pasos. ¿Qué tal tres pasos?

¡Ajá! Ahora el Puente Golden Gate aparece. Así que toma tres pasos llegar de Twin Peaks al puente usando esta ruta.

Hay otras rutas que te llevarán al puente también, pero son más largas (cuatro pasos). El algoritmo encontró que la ruta más corta al puente tiene tres pasos de largo. Este tipo de problema se llama un problema de camino más corto. Siempre estás tratando de encontrar el algo más corto. Podría ser la ruta más corta a la casa de tu amigo. O tal vez estés navegando la web. Sin que lo sepas, la red está buscando el camino más corto entre tu computadora y el servidor de un sitio web. El algoritmo para resolver un problema de camino más corto se llama búsqueda en anchura.

Para averiguar cómo llegar de Twin Peaks al Puente Golden Gate, hay dos pasos:

1. Modelar el problema como un grafo.
2. Resolver el problema usando búsqueda en anchura.

A continuación cubriré qué son los grafos. Luego entraré en búsqueda en anchura en más detalle.

**¿Qué es un grafo?**

Un grafo modela un conjunto de conexiones. Por ejemplo, supón que tú y tus amigos están jugando póker, y quieres modelar quién le debe dinero a quién. Así es como podrías decir, "Alex le debe dinero a Rama."

El grafo completo podría verse algo así.

**Grafo de personas que le deben dinero a otras personas en póker**

Alex le debe dinero a Rama, Tom le debe dinero a Adit, y así sucesivamente.

Cada grafo está hecho de nodos y bordes.

¡Eso es todo! Los grafos están hechos de nodos y bordes. Un nodo puede estar directamente conectado a muchos otros nodos. Esos nodos se llaman sus vecinos de entrada o vecinos de salida.

Como Alex está apuntando a Rama, Alex es vecino de entrada de Rama (y Rama es vecino de salida de Alex). Esta terminología puede ser confusa, así que aquí hay un diagrama para ayudar:

En este grafo, Adit no es vecino de entrada o de salida de Alex, porque no están directamente conectados. Pero Adit es vecino de salida de Rama y Tom.

Los grafos son una forma de modelar cómo diferentes cosas están conectadas entre sí. Ahora veamos búsqueda en anchura en acción.

**Búsqueda en anchura**

Miramos un algoritmo de búsqueda en el capítulo 1: búsqueda binaria. La búsqueda en anchura es un tipo diferente de algoritmo de búsqueda: uno que se ejecuta en grafos. Puede ayudar a responder dos tipos de preguntas:

• **Tipo de pregunta 1:** ¿Hay un camino del nodo A al nodo B? • **Tipo de pregunta 2:** ¿Cuál es el camino más corto del nodo A al nodo B?

Ya viste búsqueda en anchura una vez, cuando calculaste la ruta más corta de Twin Peaks al Puente Golden Gate. Esa era una pregunta de tipo 2: "¿Cuál es el camino más corto?" Ahora veamos el algoritmo en más detalle. Harás una pregunta de tipo 1: "¿Hay un camino?"

Supón que eres el orgulloso dueño de una granja de mangos. Estás buscando un vendedor de mangos que pueda vender tus mangos. ¿Estás conectado a un vendedor de mangos en Facebook? Bueno, puedes buscar a través de tus amigos.

Esta búsqueda es bastante directa. Primero, haz una lista de amigos para buscar.

Ahora, ve a cada persona en la lista y verifica si esa persona vende mangos.

Supón que ninguno de tus amigos son vendedores de mangos. Ahora tienes que buscar a través de los amigos de tus amigos.

Cada vez que buscas a alguien de la lista, agrega a todos sus amigos a la lista también.

De esta manera, no solo buscas tus amigos, sino que buscas sus amigos también. Recuerda, el objetivo es encontrar un vendedor de mangos en tu red. Así que si Alice no es vendedora de mangos, agregas sus amigos a la lista también. Eso significa que eventualmente buscarás sus amigos—y luego sus amigos, y así sucesivamente. Con este algoritmo, buscarás toda tu red hasta que encuentres un vendedor de mangos. Esta algoritmo es búsqueda en anchura.

**Encontrando el camino más corto**

Como resumen, estas son las dos preguntas que la búsqueda en anchura puede responder para ti:

• **Tipo de pregunta 1:** ¿Hay un camino del nodo A al nodo B? (¿Hay un vendedor de mangos en tu red?) • **Tipo de pregunta 2:** ¿Cuál es el camino más corto del nodo A al nodo B? (¿Quién es el vendedor de mangos más cercano?)

Viste cómo responder la pregunta 1; ahora tratemos de responder la pregunta 2. ¿Puedes encontrar el vendedor de mangos más cercano? Por ejemplo, tus amigos son conexiones de primer grado, y sus amigos son conexiones de segundo grado.

Preferirías una conexión de primer grado a una de segundo grado, y una conexión de segundo grado a una de tercer grado, y así sucesivamente. Así que no deberías buscar ninguna conexión de segundo grado antes de asegurarte de que no tienes una conexión de primer grado que sea vendedor de mangos. Bueno, ¡la búsqueda en anchura ya hace esto! La forma en que funciona la búsqueda en anchura, la búsqueda se irradia desde el punto de partida. Así que verificarás conexiones de primer grado antes que conexiones de segundo grado. Pregunta rápida: ¿quién será verificado primero, Claire o Anuj? Respuesta: Claire es una conexión de primer grado, y Anuj es una conexión de segundo grado. Así que Claire será verificada antes que Anuj.

Otra forma de ver esto es, las conexiones de primer grado se agregan a la lista de búsqueda antes que las conexiones de segundo grado.

Simplemente bajas por la lista y verificas personas para ver si cada una es vendedora de mangos. Las conexiones de primer grado serán buscadas antes que las conexiones de segundo grado, así que encontrarás el vendedor de mangos más cercano a ti. La búsqueda en anchura no solo encuentra un camino de A a B, también encuentra el camino más corto.

Nota que esto solo funciona si buscas personas en el mismo orden en que fueron agregadas. Es decir, si Claire fue agregada a la lista antes que Anuj, Claire necesita ser buscada antes que Anuj. ¿Qué pasa si buscas Anuj antes que Claire, y ambos son vendedores de mangos? Bueno, Anuj es un contacto de segundo grado, y Claire es un contacto de primer grado. Terminas con un vendedor de mangos que no es el más cercano a ti en tu red. Así que necesitas buscar personas en el orden en que fueron agregadas. Hay una estructura de datos para esto: se llama cola.

**Colas**

Una cola funciona exactamente como lo hace en la vida real. Supón que tú y tu amigo están haciendo cola en la parada del autobús. Si estás antes que él en la cola, subes al autobús primero. Una cola funciona de la misma manera. Las colas son similares a las pilas. No puedes acceder a elementos aleatorios en la cola. En su lugar, solo hay dos operaciones, enqueue y dequeue.

Si haces enqueue de dos elementos a la lista, el primer elemento que agregaste será dequeued antes que el segundo elemento. Puedes usar esto para tu lista de búsqueda! Las personas que se agregan a la lista primero serán dequeued y buscadas primero.

La cola se llama una estructura de datos FIFO: First In, First Out (Primero en Entrar, Primero en Salir). En contraste, una pila es una estructura de datos LIFO: Last In, First Out (Último en Entrar, Primero en Salir).

Ahora que sabes cómo funciona una cola, ¡implementemos búsqueda en anchura!

**Ejercicios**

Ejecuta el algoritmo de búsqueda en anchura en cada uno de estos grafos para encontrar la solución.

1. **6.1** Encuentra la longitud del camino más corto de start a finish.
2. **6.2** Encuentra la longitud del camino más corto de "cab" a "bat".

**Implementando el grafo**

Primero, necesitas implementar el grafo en código. Un grafo consiste en varios nodos.

Y cada nodo está conectado a otros nodos. ¿Cómo expresas una relación como "tú -> bob"? Afortunadamente, conoces una estructura de datos que te permite expresar relaciones: ¡una tabla hash!

Recuerda, una tabla hash te permite mapear una clave a un valor. En este caso, quieres mapear un nodo a todos sus vecinos de salida.

Así es como lo escribirías en Python:

graph = {}

graph["you"] = ["alice", "bob", "claire"]

Nota que "you" está mapeado a un array. Así que graph["you"] te dará un array de todos los vecinos de salida de "you". Recuerda que los vecinos de salida son los nodos a los que apunta el nodo "you".

Un grafo es solo un montón de nodos y bordes, ¡así que esto es todo lo que necesitas para tener un grafo en Python! ¿Qué pasa con un grafo más grande, como este?

Aquí está como código Python:

graph = {}

graph["you"] = ["alice", "bob", "claire"]

graph["bob"] = ["anuj", "peggy"]

graph["alice"] = ["peggy"]

graph["claire"] = ["thom", "jonny"]

graph["anuj"] = []

graph["peggy"] = []

graph["thom"] = []

graph["jonny"] = []

Pregunta rápida: ¿importa en qué orden agregues los pares clave/valor? ¿Importa si escribes

graph["claire"] = ["thom", "jonny"]

graph["anuj"] = []

en lugar de

graph["anuj"] = []

graph["claire"] = ["thom", "jonny"]

Piensa en el capítulo anterior. Respuesta: No importa. Las tablas hash no tienen ordenamiento, así que no importa en qué orden agregues pares clave/valor.

Anuj, Peggy, Thom, y Jonny no tienen vecinos de salida. Tienen vecinos de entrada, ya que tienen flechas apuntando hacia ellos—pero no hay flechas de ellos hacia alguien más. Esto se llama un grafo dirigido: la relación es solo en una dirección. Un grafo no dirigido no tiene flechas. Por ejemplo, ambos grafos son iguales.

Si tienes un grafo no dirigido, puedes olvidar los términos vecino de entrada y vecino de salida, y usar el término más simple vecino.

**Implementando el algoritmo**

Para recapitular, así es como funcionará la implementación.

**NOTA** Cuando actualizo colas, uso los términos enqueue y dequeue. También encontrarás los términos push y pop. Push es casi siempre lo mismo que enqueue, y pop es casi siempre lo mismo que dequeue.

Haz una cola para comenzar. En Python, usas la función de cola de dos extremos (deque) para esto:

from collections import deque

search\_queue = deque() #A

search\_queue += graph["you"] #B

#A Crea una nueva cola

#B Agrega todos tus vecinos de salida a la cola de búsqueda

Recuerda, graph["you"] te dará una lista de todos tus vecinos de salida, como ["alice", "bob", "claire"]. Todos esos se agregan a la cola de búsqueda.

Veamos el resto:

while search\_queue: #A

person = search\_queue.popleft() #B

if person\_is\_seller(person): #C

print(person + " is a mango seller!") #D

return True

else:

search\_queue += graph[person] #E

return False #F

#A Mientras la cola no esté vacía …

#B … toma la primera persona de la cola

#C Verifica si la persona es vendedora de mangos

#D Sí, es vendedora de mangos.

#E No, no lo es. Agrega todos los amigos de esta persona a la cola de búsqueda.

#F Si llegaste aquí, nadie en la cola era vendedor de mangos.

Una cosa final: aún necesitas una función person\_is\_seller para decirte cuándo alguien es vendedor de mangos. Aquí hay una:

def person\_is\_seller(name):

return name[-1] == 'm'

Esta función verifica si el nombre de la persona termina con la letra m. Si lo hace, es vendedor de mangos. Una forma medio tonta de hacerlo, ¡pero servirá para este ejemplo! Ahora veamos la búsqueda en anchura en acción.

Y así sucesivamente. El algoritmo seguirá hasta que:

• Se encuentre un vendedor de mangos, o • La cola se vuelva vacía, en cuyo caso no hay vendedor de mangos.

Alice y Bob comparten un amigo: Peggy. Así que Peggy será agregada a la cola dos veces: una vez cuando agregues los amigos de Alice, y de nuevo cuando agregues los amigos de Bob. Terminarás con dos Peggys en la cola de búsqueda.

Pero solo necesitas verificar Peggy una vez para ver si es vendedora de mangos. Si la verificas dos veces, estás haciendo trabajo innecesario extra. Así que una vez que busques una persona, deberías marcar a esa persona como buscada y no buscarla de nuevo.

Si no haces esto, también podrías terminar en un bucle infinito. Supón que el grafo de vendedor de mangos se veía así.

Para comenzar, la cola de búsqueda contiene todos tus vecinos de salida.

Ahora verificas Peggy. Ella no es vendedora de mangos, así que agregas todos sus vecinos de salida a la cola de búsqueda.

A continuación, te verificas a ti mismo. No eres vendedor de mangos, así que agregas todos tus vecinos de salida a la cola de búsqueda.

Y así sucesivamente. Esto será un bucle infinito, porque la cola de búsqueda seguirá yendo de ti a Peggy.

Antes de verificar una persona, es importante asegurarse de que no hayan sido verificadas ya. Para hacer eso, mantendrás una lista de personas que ya has verificado.

Aquí está el código final para búsqueda en anchura, tomando eso en cuenta:

def search(name):

search\_queue = deque()

search\_queue += graph[name]

searched = set() #A

while search\_queue:

person = search\_queue.popleft()

if not person in searched: #B

if person\_is\_seller(person):

print(person + " is a mango seller!")

return True

else:

search\_queue += graph[person]

searched.add(person) #C

return False

search("you")

#A Este conjunto es cómo mantienes seguimiento de qué personas has buscado antes.

#B Solo busca esta persona si no la has buscado ya.

#C Marca esta persona como buscada

Trata de ejecutar este código tú mismo. Tal vez trata de cambiar la función person\_is\_seller a algo más significativo, y ve si imprime lo que esperas.

**Tiempo de ejecución**

Si buscas toda tu red para un vendedor de mangos, eso significa que seguirás cada borde (recuerda, un borde es la flecha o conexión de una persona a otra). Así que el tiempo de ejecución es al menos O(número de bordes).

También mantienes una cola de cada persona para buscar. Agregar una persona a la cola toma tiempo constante: O(1). Hacer esto para cada persona tomará O(número de personas) en total. La búsqueda en anchura toma O(número de personas + número de bordes), y se escribe más comúnmente como O(V+E) (V para número de vértices, E para número de bordes).

**Ejercicio**

Aquí hay un pequeño grafo de mi rutina matutina.

Te dice que no puedo desayunar hasta que me haya lavado los dientes. Así que "desayunar" depende de "lavarse los dientes".

Por otro lado, ducharme no depende de lavarme los dientes, porque puedo ducharme antes de lavarme los dientes. De este grafo, puedes hacer una lista del orden en que necesito hacer mi rutina matutina:

1. Despertarse.
2. Ducharse.
3. Lavarse los dientes.
4. Desayunar.

Nota que "ducharse" puede moverse, así que esta lista también es válida:

1. Despertarse.
2. Lavarse los dientes.
3. Ducharse.
4. Desayunar.
5. **6.3** Para estas tres listas, marca si cada una es válida o inválida.
6. **6.4** Aquí hay un grafo más grande. Haz una lista válida para este grafo.

Podrías decir que esta lista está ordenada, de cierta manera. Si la tarea A depende de la tarea B, la tarea A aparece más tarde en la lista. Esto se llama un ordenamiento topológico, y es una forma de hacer una lista ordenada de un grafo. Supón que estás planeando una boda y tienes un grafo grande lleno de tareas por hacer—y no estás seguro por dónde empezar. Podrías ordenar topológicamente el grafo y obtener una lista de tareas por hacer, en orden.

Supón que tienes un árbol genealógico.

Este es un grafo, porque tienes nodos (las personas) y bordes.

Los bordes apuntan a los nodos padres. Pero todos los bordes van hacia abajo—¡no tendría sentido que un árbol genealógico tuviera un borde apuntando hacia atrás! Eso sería sin sentido—¡tu papá no puede ser el papá de tu abuelo!

Esto se llama un árbol. Un árbol es un tipo especial de grafo, donde ningún borde apunta hacia atrás.

1. **6.5** ¿Cuáles de los siguientes grafos también son árboles?

**Resumen**

• La búsqueda en anchura te dice si hay un camino de A a B. • Si hay un camino, la búsqueda en anchura encontrará el camino más corto. • Si tienes un problema como "encuentra el X más corto," trata de modelar tu problema como un grafo, y usa búsqueda en anchura para resolverlo. • Un grafo dirigido tiene flechas, y la relación sigue la dirección de la flecha (rama -> adit significa "rama le debe dinero a adit"). • Los grafos no dirigidos no tienen flechas, y la relación va en ambas direcciones (ross - rachel significa "ross salió con rachel y rachel salió con ross"). • Las colas son FIFO (First In, First Out). • Las pilas son LIFO (Last In, First Out). • Necesitas verificar personas en el orden en que fueron agregadas a la lista de búsqueda, así que la lista de búsqueda necesita ser una cola. De lo contrario, no obtendrás el camino más corto. • Una vez que verificas alguien, asegúrate de no verificarlo de nuevo. De lo contrario, podrías terminar en un bucle infinito.

**Capítulo 7: Árboles**

**En este capítulo**

• Aprendes qué es un árbol, y la diferencia entre árboles y grafos • Te sientes cómodo ejecutando un algoritmo sobre un árbol • Aprendes búsqueda en profundidad, y ves la diferencia entre búsqueda en profundidad y búsqueda en anchura • Aprendes codificación de Huffman, un algoritmo de compresión que hace uso de árboles

¿Qué tienen en común los algoritmos de compresión y el almacenamiento de bases de datos? A menudo hay un árbol debajo haciendo todo el trabajo duro. Los árboles son un subconjunto de grafos. Vale la pena cubrirlos por separado ya que hay muchos tipos especializados de árboles. Por ejemplo, la codificación de Huffman, que es un algoritmo de compresión que aprenderás en este capítulo, usa árboles binarios. La mayoría de las bases de datos por otro lado, usan un árbol balanceado como un árbol B, que aprenderás en el siguiente capítulo.

Hay tantos tipos de árboles por ahí. Estos dos capítulos te darán el vocabulario y conceptos que necesitas para entenderlos.

**Tu primer árbol**

Los árboles son un tipo de grafo. Tendremos una definición más completa más tarde. Primero aprendamos algo de terminología y veamos un ejemplo.

Al igual que los grafos, los árboles están hechos de nodos y bordes:

En este libro, trabajaremos con árboles con raíz. Los árboles con raíz tienen un nodo que lleva a todos los otros nodos:

Trabajaremos exclusivamente con árboles con raíz, así que cuando digo "árbol" en este capítulo, me refiero a un árbol con raíz. Los nodos pueden tener hijos, y los nodos hijos pueden tener un padre:

En un árbol, los nodos tienen como máximo un padre. El único nodo sin padres es la raíz. Los nodos sin hijos se llaman nodos hoja:

Si entiendes raíz, hoja, padre, hijo, ¡estás listo para continuar leyendo!

**Directorios de archivos**

Como un árbol es un tipo de grafo, podemos ejecutar un algoritmo de grafo en él. En el capítulo 6, habíamos conocido búsqueda en anchura, un algoritmo para encontrar el camino más corto en un grafo. Vamos a usar búsqueda en anchura en un árbol. Si no te sientes cómodo con búsqueda en anchura, echa un vistazo al capítulo 6.

Un directorio de archivos es un árbol con el que todos interactuamos todos los días. Supón que tengo este directorio de archivos:

Quiero imprimir el nombre de cada archivo en el directorio pics, incluyendo todos sus subdirectorios. Aquí solo hay un subdirectorio, 2001. Podemos usar búsqueda en anchura para hacer esto! Primero, déjame mostrarte cómo se ve este directorio de archivos como un árbol:

Como este directorio de archivos es un árbol, podemos ejecutar un algoritmo de grafo en él. Habíamos usado búsqueda en anchura anteriormente como un algoritmo de búsqueda. Pero la búsqueda no es lo único para lo que es buena. La búsqueda en anchura es un algoritmo de recorrido. Eso significa que es un algoritmo que visita cada nodo en un árbol, también conocido como recorrer o caminar el árbol. ¡Eso es exactamente lo que necesitamos! Necesitamos un algoritmo que vaya a cada archivo en este árbol e imprima su nombre. El algoritmo también entrará en subdirectorios, encontrará archivos allí, e imprimirá sus nombres.

Mi lógica será:

1. Visitar cada nodo en el árbol.
2. Si este nodo es un archivo, imprimir su nombre.
3. Si el nodo es una carpeta, agregarlo a una cola de carpetas para buscar archivos.

El código está abajo. Es muy similar al código del vendedor de mangos del capítulo 6:

from os import listdir

from os.path import isfile, join

from collections import deque

def printnames(start\_dir):

# usamos una cola para mantener seguimiento de

# carpetas para buscar

search\_queue = deque()

search\_queue.append(start\_dir)

# mientras la cola no esté vacía,

# quita una carpeta para revisar

while search\_queue:

dir = search\_queue.popleft()

# repasa cada archivo y carpeta en esta carpeta

for file in sorted(listdir(dir)):

fullpath = join(dir, file)

if isfile(fullpath):

# si es un archivo, imprime el nombre

print(file)

else:

# si es una carpeta,

# agrégala a la cola de carpetas para buscar

search\_queue.append(fullpath)

printnames("pics")

Aquí usamos una cola como lo hicimos en el ejemplo del vendedor de mangos. En la cola mantenemos seguimiento de qué carpetas aún necesitamos buscar. Por supuesto, en ese ejemplo paramos una vez que encontramos un vendedor de mangos, pero aquí recorremos todo el árbol.

Hay otra diferencia importante del código del vendedor de mangos. ¿Puedes detectarla?

Recuerda cómo en el ejemplo del vendedor de mangos, tuvimos que mantener seguimiento de si ya habíamos buscado una persona:

# Solo busca esta persona si no la has buscado ya.

if person not in searched:

if person\_is\_seller(person):

...

¡No tenemos que hacer eso aquí! Los árboles no tienen ciclos, y cada nodo solo tiene un padre. No hay forma de que accidentalmente busquemos la misma carpeta más de una vez, o terminemos en un bucle infinito, así que no hay necesidad de mantener seguimiento de qué carpetas ya hemos buscado. Simplemente no hay una forma de revisitar una carpeta.

Esta propiedad de los árboles ha hecho nuestro código más simple. Esa es una conclusión importante de este capítulo: los árboles no tienen ciclos.

**Una nota sobre enlaces simbólicos**

Puedes saber qué son los enlaces simbólicos. Si no lo sabes, los enlaces simbólicos son una forma de introducir un ciclo en un directorio de archivos. Así es como podría hacer un enlace simbólico en macOS o Linux:

ln -s pics/ pics/2001/pics

O en Windows:

mklink /d pics/ pics/2001/pics

Si hiciera eso, el árbol se vería así:

¡Ahora nuestro directorio de archivos ya no es un árbol! Para mantener las cosas simples para este ejemplo, vamos a ignorar los enlaces simbólicos. Si tuviéramos un enlace simbólico, Python es lo suficientemente inteligente para evitar un bucle infinito. Aquí está el error que lanza:

OSError: [Errno 62] Too many levels of symbolic links: 'pics/2001/pics'

**Una Odisea del Espacio: búsqueda en profundidad**

Recorramos nuestro directorio de archivos de nuevo, haciéndolo recursivamente esta vez:

from os import listdir

from os.path import isfile, join

def printnames(dir):

# repasa cada archivo y carpeta en la carpeta actual

for file in sorted(listdir(dir)):

fullpath = join(dir, file)

if isfile(fullpath):

# si es un archivo, imprime el nombre

print(file)

else:

# si es una carpeta, llama esta función recursivamente en ella

# para buscar archivos y carpetas

printnames(fullpath)

printnames("pics")

Nota que ahora no estamos usando una cola. En su lugar, cuando encontramos una carpeta, inmediatamente miramos dentro para más archivos y carpetas.

Ahora tenemos dos formas de listar los nombres de archivos. ¡Pero aquí está la parte sorprendente: ambas soluciones imprimirán los nombres de archivos en órdenes diferentes!

Una imprime los nombres así:

a.png

space.png

odyssey.png

La otra imprime esto:

odyssey.png

a.png

space.png

¿Puedes averiguar qué solución imprime qué orden, y por qué? Pruébalo tú mismo antes de continuar.

La primera solución usa búsqueda en anchura. Cuando encuentra una carpeta, esa carpeta se agrega a la cola, para ser verificada más tarde. Así que el algoritmo va a la carpeta 2001/, no entra en ella pero la agrega a la cola para ser revisada más tarde, imprime todos los nombres de archivos en la carpeta pics/, luego vuelve a la carpeta 2001/ e imprime los nombres de archivos allí:

Puedes ver que el algoritmo visita la carpeta 2001 primero, pero no mira adentro. Esa carpeta se agrega solo a la cola, y la búsqueda en anchura se mueve a odyssey.png.

La segunda solución usa un algoritmo llamado búsqueda en profundidad. La búsqueda en profundidad también es un algoritmo de recorrido de grafos y árboles. Cuando encuentra una carpeta, mira dentro inmediatamente en lugar de agregarla a una cola:

La segunda solución es la que imprime:

a.png

space.png

odyssey.png

La búsqueda en anchura y la búsqueda en profundidad están estrechamente relacionadas, y a menudo donde se menciona una, también se mencionará la otra. Ambos algoritmos imprimieron todos los nombres de archivos, así que ambos funcionan para este ejemplo. Pero hay una gran diferencia. ¡La búsqueda en profundidad no puede usarse para encontrar el camino más corto!

En el ejemplo del vendedor de mangos, no podríamos haber usado búsqueda en profundidad. Dependemos del hecho de que estamos verificando todos nuestros amigos de primer grado antes que los amigos de segundo grado, y así sucesivamente. Eso es lo que hace la búsqueda en anchura. Pero la búsqueda en profundidad irá tan profundo como sea posible de inmediato. Puede encontrarte un vendedor de mangos a tres grados de distancia, ¡cuando tienes un contacto más cercano! Supón que esta es tu red social:

Digamos que procesamos nodos en orden de izquierda a derecha. La búsqueda en profundidad llegará al nodo hijo más a la izquierda e irá profundo:

Como la búsqueda en profundidad fue profunda en el nodo izquierdo, falló en darse cuenta de que el nodo derecho es un vendedor de mangos que está mucho más cerca.

La búsqueda en anchura encontrará correctamente el vendedor de mangos más cercano:

Así que mientras ambos algoritmos funcionaron para listar archivos, solo la búsqueda en anchura funciona para encontrar el camino más corto. La búsqueda en profundidad tiene otros usos. Puede usarse para encontrar el ordenamiento topológico, un concepto que vimos brevemente en el capítulo 6.

**Una mejor definición de árboles**

Ahora que has visto un ejemplo, es hora de una mejor definición de un árbol. Un árbol es un grafo conectado y acíclico:

Como había dicho antes, estamos trabajando exclusivamente con árboles con raíz, así que nuestros árboles todos tienen una raíz también. Y estamos trabajando exclusivamente con grafos conectados. Así que lo más importante para recordar es: los árboles no pueden tener ciclos.

Ahora hemos visto un árbol en acción, enfoquémonos en un tipo específico de árbol.

**Árboles binarios**

Las ciencias de la computación están llenas de diferentes tipos de árboles. Los árboles binarios son un tipo muy común de árbol. Para el resto de este capítulo, y la mayoría del siguiente, trabajaremos con árboles binarios.

Un árbol binario es un tipo especial de árbol donde los nodos pueden tener como máximo dos hijos (de ahí el nombre "binario", que significa "dos"). Estos se llaman tradicionalmente hijo izquierdo e hijo derecho:

Un árbol genealógico es un ejemplo de un árbol binario, ya que todos tienen dos padres biológicos:

En ese ejemplo hay una conexión clara entre nodos—todos son familia. Pero los datos pueden ser totalmente arbitrarios:

Lo importante es que nunca tengas más de dos hijos. A veces también verás gente refiriéndose al subárbol izquierdo o subárbol derecho:

Los árboles binarios aparecen en todas partes en las ciencias de la computación. Vamos a pasar el resto de este capítulo viendo un ejemplo que usa un árbol binario.

**Codificación de Huffman**

La Codificación de Huffman es un ejemplo ordenado de usar árboles binarios. También es la base para algoritmos de compresión de texto. No describiremos el algoritmo, pero pasaremos tiempo enfocándonos en cómo funciona, y cómo hace uso inteligente de árboles.

Primero, un poco de trasfondo. Para saber cómo funciona la compresión, necesitamos saber cuánto espacio toma un archivo de texto. Supón que tenemos un archivo de texto con solo una palabra: "tilt". ¿Cuánto espacio usa eso? Puedes usar el comando stat (disponible en Unix). Primero, guarda la palabra en un archivo llamado test.txt. Luego, usando stat:

$ cat test.txt

tilt

$ stat -f%z test.txt

4

Así que ese archivo ocupa cuatro bytes: uno por carácter.

Esto tiene sentido. Asumiendo que estamos usando ISO-8859-1 (ve la barra lateral para lo que esto significa), cada letra ocupa exactamente un byte. Por ejemplo, la letra a es código ISO-8859-1 97, que puedo escribir en binario como: 01100001. Eso es ocho bits. Un bit es un dígito que puede ser cero o uno. Y hay ocho de ellos. Ocho bits es un byte. Así que la letra a se representa usando un byte. El código ISO-8859-1 va de 00000000, que representa el carácter nulo, hasta 11111111, que representa ÿ ("letra pequeña latina y con diéresis"). Hay 256 combinaciones posibles de ceros y unos con ocho bits, así que el código ISO-8859-1 permite 256 letras posibles.

**Codificación de caracteres**

Como este ejemplo te mostrará, hay muchas formas diferentes de codificar caracteres. Es decir, la letra a podría escribirse en binario de muchas formas diferentes.

Comenzó con ASCII. En los 1960s, se creó ASCII. ASCII es una codificación de 7 bits. Desafortunadamente, ASCII no incluía muchos caracteres. ASCII no incluye ningún carácter con diéresis (ü o ö por ejemplo), o monedas comunes como la libra británica o el yen japonés.

Así que se creó ISO-8859-1. ISO-8859-1 es una codificación de 8 bits, así que duplica el número de caracteres que ASCII proporcionó. Pasamos de 128 caracteres a 256 caracteres. Pero esto aún no era suficiente, y los países comenzaron a hacer sus propias codificaciones. Por ejemplo, Japón tiene varias codificaciones para japonés, ya que ISO-8859-1 y ASCII estaban enfocados en idiomas europeos. Toda la situación era un desastre hasta que se introdujo Unicode.

Unicode es un estándar de codificación. Apunta a proporcionar caracteres para cualquier idioma. Unicode tiene 149,186 caracteres a partir de 2022—¡quite un salto desde 256! Más de 1000 de estos son emojis.

Unicode es el estándar, pero necesitas usar una codificación que siga el estándar. La codificación más popular hoy es UTF-8. UTF-8 es codificación de caracteres de longitud variable, lo que significa que los caracteres pueden ser de uno a cuatro bytes (8 a 32 bits).

No necesitas preocuparte demasiado por esto. He mantenido el ejemplo simple intencionalmente usando ISO-8859-1, que es 8 bits, que es una cantidad consistente y agradable de bits para trabajar.

Solo recuerda estas conclusiones: • Los algoritmos de compresión tratan de reducir el número de bits necesarios para almacenar cada carácter. • Si necesitas elegir una codificación para un proyecto, UTF-8 es una buena elección por defecto.

Decodifiquemos algo de binario a ISO-8859-1 juntos: 011100100110000101100100. Puedes buscar en Google una tabla ISO-8859-1 o un convertidor de binario a ISO-8859-1 para hacer esto más fácil.

Primero, sabemos que cada letra es ocho bits, así que voy a dividir esto en trozos de ocho, para hacer más fácil leerlo: 01110010 01100001 01100100

Genial, ahora vemos que hay tres letras. Buscándolas en una tabla ISO-8859-1, veo que deletrean rad: 01110010 es r, y así sucesivamente. Así es como tu editor de texto toma los datos binarios en un archivo de texto y los muestra como ISO-8859-1. Puedes ver la información binaria usando xxd. Esta utilidad está disponible en Unix. Así es como tilt se ve en binario:

$ xxd -b test.txt

00000000: 01110100 01101001 01101100 01110100 tilt

Aquí es donde entra la compresión. Para la palabra "tilt", no necesitamos 256 letras posibles; solo necesitamos tres. Así que no necesitamos ocho bits, solo necesitamos dos. Podríamos crear nuestro propio código de dos bits, solo para estas tres letras:

t = 00

i = 01

l = 10

Así es como podríamos escribir tilt usando nuestro nuevo código: 00011000. Puedo hacer esto más fácil de leer agregando espacios de nuevo: 00 01 10 00. Si lo comparas con el mapeo arriba, verás que esto deletrea tilt.

Esto es lo que hace la Codificación de Huffman—mira los caracteres que se están usando y trata de usar menos de ocho bits. Así es como comprime los datos. La Codificación de Huffman genera un árbol:

Puedes usar este árbol para encontrar el código para cada letra. Comenzando en el nodo raíz, encuentra un camino hacia abajo a la letra L. Cada vez que elijas una rama izquierda, agrega un cero a tu código. Cuando elijas una rama derecha, agrega uno. Cuando llegues a una letra, para de progresar por el árbol. Así que el código para la letra L es 01. Aquí están los tres códigos dados por el árbol:

i = 00

l = 01

t = 1

Nota que la letra T tiene un código de solo un dígito. A diferencia de ISO-8859-1, en la Codificación de Huffman, los códigos no todos tienen que ser de la misma longitud. Esto es importante—veamos otro ejemplo para entender por qué.

Queremos ahora comprimir la frase "paranoid android". Aquí está el árbol generado por el algoritmo de Codificación de Huffman:

Verifícate: ¿cuál es el código para la letra P? Pruébalo tú mismo antes de leer. Es 0001. ¿Qué tal la letra D? Es 10.

En este caso, ¡en realidad hay tres longitudes diferentes posibles! Supón que tratamos de decodificar algunos datos binarios: 01101010. Vemos el problema de inmediato: ¡no podemos trocearlo como lo hicimos con ISO-8859-1! Mientras que todos los códigos ISO-8859-1 eran ocho dígitos, aquí el código podría ser 2, 3, o 4 dígitos. Como la longitud del código varía, no podemos usar troceado más.

En su lugar, necesitamos mirar un dígito a la vez, como si estuviéramos mirando una cinta. Así es como hacerlo: el primer número es cero, así que ve a la izquierda (solo estoy mostrando parte del árbol aquí):

Luego obtenemos un uno, así que ve a la derecha:

Luego otro uno, así que derecha de nuevo:

¡Ajá! Encontramos una letra. Estos son los datos binarios que nos quedan: 01010. Podemos comenzar de nuevo en el nodo raíz y encontrar las otras letras. Trata de decodificar el resto tú mismo y luego lee. ¿Obtuviste la palabra? Era rad. Esta es una gran diferencia entre la Codificación de Huffman e ISO-8859-1. Los códigos pueden variar, así que la decodificación necesita hacerse de manera diferente.

Es más trabajo hacerlo de esta manera en lugar de troceado. Pero hay un gran beneficio. Nota que las letras que aparecen más a menudo, tienen códigos más cortos. D aparece tres veces así que su código es solo dos dígitos, versus I que aparece dos veces, y P que aparece solo una vez. En lugar de asignar cuatro bits a todo, podemos comprimir letras usadas frecuentemente aún más. ¡Puedes ver cómo, en un texto más largo, esto sería un gran ahorro!

Ahora que entendemos a alto nivel cómo funciona la Codificación de Huffman, veamos qué propiedades de los árboles está aprovechando Huffman aquí.

Primero que todo, ¿podría haber superposición entre códigos? Toma este código por ejemplo:

a = 0

b = 1

c = 00

Ahora si ves el binario 001, ¿es eso aab o cb? c y a comparten parte de su código, así que no está claro. Aquí está cómo se vería el árbol para este código:

Pasamos A en el camino a C, lo que causa el problema. Eso no es un problema con la Codificación de Huffman, porque las letras solo aparecen en nodos hoja. Y hay un camino único de la raíz a cada nodo hoja—esa es una de las propiedades de los árboles. Así que podemos garantizar que la superposición no es un problema.

Esto también garantiza que solo hay un código para cada letra. Tener múltiples caminos a cada letra significaría que hay múltiples códigos asignados a cada letra, lo que sería innecesario.

Cuando leemos el código un dígito a nuestro tiempo, estamos asumiendo que eventualmente terminaremos en una letra. Si esto fuera un grafo con un ciclo, no podríamos hacer esa suposición. Podríamos quedarnos atrapados en el ciclo y terminar en un bucle infinito:

Pero como esto es un árbol, sabemos que no hay ciclos, y así estamos garantizados de terminar en alguna letra.

Estamos usando un árbol con raíz. Los árboles con raíz tienen un nodo raíz, lo que es importante porque ¡necesitamos saber dónde empezar! Los grafos no necesariamente tienen un nodo raíz.

Finalmente, el tipo de árbol usado aquí se llama un árbol binario. Los árboles binarios pueden tener como máximo dos hijos—el hijo izquierdo y el hijo derecho. Esto tiene sentido porque el binario solo tiene dos dígitos. Si hubiera un tercer hijo, no estaría claro qué dígito se supone que represente.

Este capítulo te introdujo a los árboles. En el siguiente capítulo, veremos algunos tipos diferentes de árboles y para qué se usan.

**Resumen**

• Los árboles son un tipo de grafo, pero los árboles no tienen ciclos. • La búsqueda en profundidad es otro algoritmo de recorrido de grafos. No puede usarse para encontrar caminos más cortos. • Un árbol binario es un tipo especial de árbol donde los nodos pueden tener como máximo dos hijos. • Hay muchos tipos diferentes de codificaciones de caracteres. Unicode es el estándar internacional, y UTF-8 es la codificación Unicode más común.