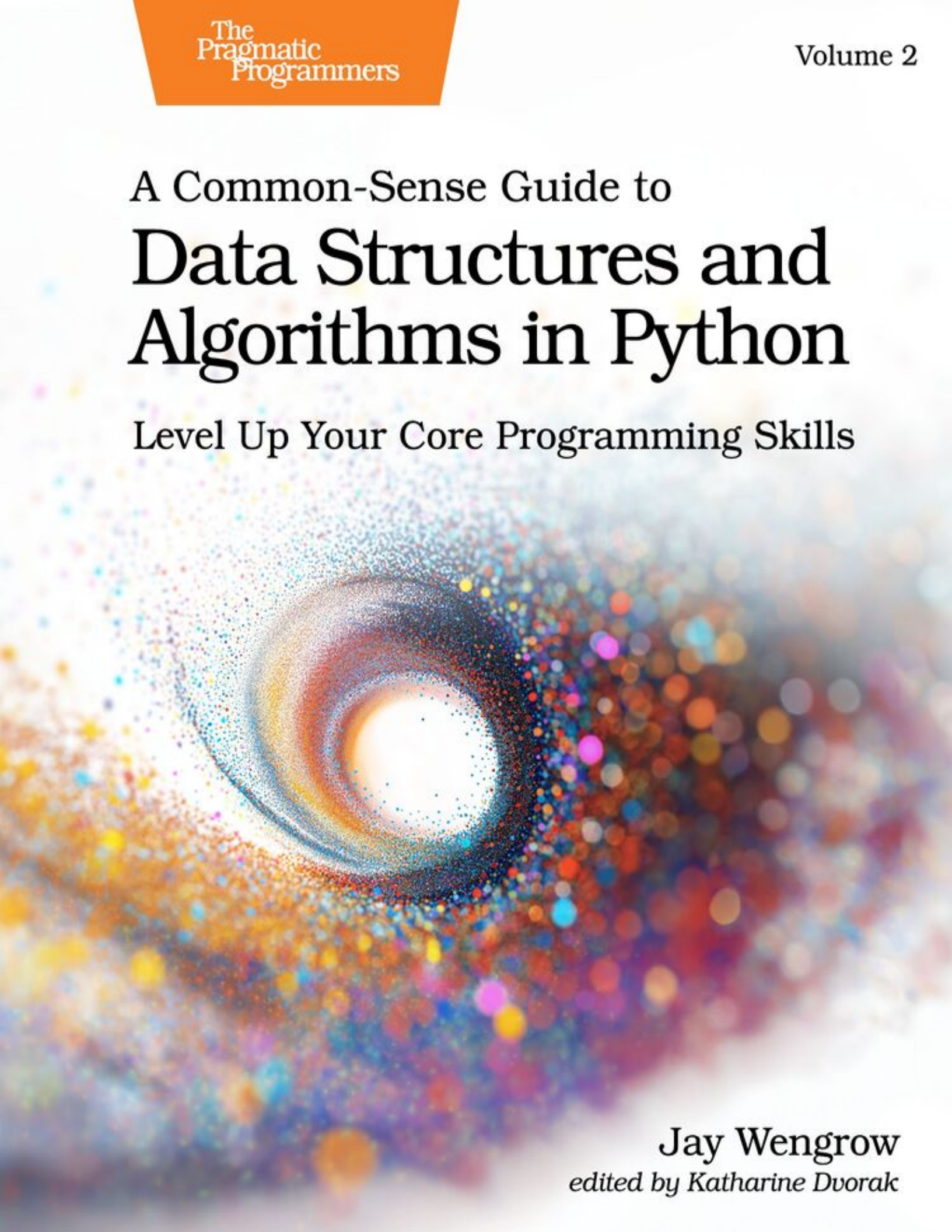
The Pragmatic Programmers

Volume 2

A Common-Sense Guide to

Data Structures and Algorithms in Python

Level Up Your Core Programming Skills



Jay Wengrow

edited by Katharine Dvorak

A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 2

Level Up Your Core Programming Skills

by Jay Wengrow

Version: P1.0 (September 2025)

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/#about-ebooks. To learn more about this book and access the free resources, go to https://pragprog.com/book/jwpython2, the book's homepage.

Thanks for your continued support,

The Pragmatic Bookshelf

The team that produced this book includes: Dave Thomas (Publisher), Janet Furlow (COO), Susannah Davidson (Executive Editor), Katharine Dvorak (Development Editor), Karen Galle (Copy Editor), Potomac Indexing, LLC (Indexing), Gilson Graphics (Layout)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Table of Contents

Early Praise for *A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 2*

Volume 2 in this series is a comprehensive extension to the exciting common-sense approach that this notorious topic needed. If you're eager to learn more and dig deeper into how the algorithms and data structures we use daily actually work, this is the book for you!

→ Michael Pabon
Software Engineer, Discord

Jay has done it again. The second volume of *A Common-Sense Guide to Data Structures and Algorithms in Python* is every bit as good as the first (which was awesome). These are, without a doubt, the two books on the topic everyone should own.

→ Lyndon Purcell
Software Developer, OK200 Software & Apps

Authors often lose their readers by overwhelming them from the outset with details and mathematical proofs. In contrast, this book adopts an alternative pedagogical approach. Readers are first introduced to the gist of an algorithm or data structure using cleverly designed illustrations, a hallmark of Jay's work. Once the reader is adequately prepared, Jay discloses the intricate details and edge cases, laying the grounds for code implementation. This approach shines throughout the book, particularly in the chapter on red-black trees, which is a masterpiece, demonstrating how a complex subject can be taught with clarity and effectiveness.

→ Ahmad Shahba

Lead Software Engineer, Science Systems and Applications, Inc.

I am such a huge fan of this book as a self-taught developer. It takes a very tricky topic and makes it easy to understand and fun to learn.

→ Fiohann Shanahan-Dover
Software Developer

Jay sets himself apart as an excellent teacher and author by making a rather complex and obtuse subject (one that is very much needed in a software creation professional's arsenal) very approachable and enjoyable to study while still going deep.

→ Paa JAKE
Test Engineer, tech11

This book explains complex data structure concepts in a very simple and approachable way. You'll experience many "aha!" moments when you read this book.

→ Praween Kumar
Software Engineer, Cigna

Acknowledgments

Putting together a book like this is truly a team effort. I may have written the words, but it was the team who edited, reviewed, illustrated, designed, and marketed the book—plus all those who supported me throughout the entire process.

To my wife, Rena, you already knew what was involved with writing a book like this, and you *still* supported me in writing another one. Your job was much harder than mine; thank you for everything.

Thank you to my precious children: Tuvi, Leah, Shaya, Rami, Yechiel, and Kayla; thank you for just being the best kids ever. However, I still can't understand why you don't want me to read my book to you at bedtime.

To my parents, Mr. and Mrs. Howard and Debbie Wengrow, thank you for your constant support and cheerleading, as well as for having sparked my interest in computer programming in the first place. You've always been there for me every step of the way.

To my wife's parents, Mr. and Mrs. Paul and Kreindel Pinkus, thank you for your encouragement and wisdom. It always meant so much that you had my book on display in your home for such a long time.

The staff at the Pragmatic Bookshelf has always been fantastic to work with, and I can't imagine why anyone would write a book for another publishing company.

Thank you to my amazing editor, Katharine Dvorak, for taking this book to the next level. Your insightful suggestions and ideas have given this book the structure and flow that make it easy to read. I appreciate your patience and dedication in carefully reviewing each and every line of a very large, technical book.

This book wouldn't be much without its illustrations. Thank you to Chaya Donninger and Jeff Stienstra for taking my pencil sketches and turning them into beautiful visuals. Your graphic art skill and expertise really shine through. You've also been very patient with me as I've asked for multiple revisions of images as I change my mind about how they should look.

Thank you to the tech reviewers who have helped perfect this volume. I'm so appreciative of your help and have incorporated many of your suggestions into the manuscript: Connor Baskin, Chinmaya Bhondwe, Felix Bossio, Gail Eddy, Tzvi Friedman, Tamanna Grover, Sruli Herbst, Paa JAKE, Praween Kumar, Avraham Meyers, Michael Pabon, Nathan Pena, Lyndon Purcell, Cody Rutt, Abraham Sangha, Brian Schau, Ahmad Shahba, Fiohann Shanahan-Dover, Zach Siglin, Marcelo Juan Surco Salas, and Mihai Visan.

Finally, a special thank you is due to all the readers of my previous books. Because of your positive feedback, I felt confident that a sequel would be well received.

Thank you, everyone, for making this sequel a reality!

Preface

The days of slow software are long gone. People expect their technology to be fast, and I mean *really* fast. According to one study, more than half of mobile website users will abandon a site if it takes more than three seconds to load. Another study reports that for every second a web page takes to load, user satisfaction goes down by 16 percent.

At the same time, people also want their technology to fit on even the smallest of devices. Whether it's a smartphone that fits easily into your pocket or a smartwatch that fits on your wrist, people want to bring their computers wherever they go. And they don't want these devices to be any less capable than a full-fledged desktop computer. However, to get apps to work on such small devices, those apps must consume the least memory possible.

The key to writing software that's both fast and memory-efficient is the mastery of data structures and algorithms. Designing the "right" algorithm usually involves a combination of knowledge, ingenuity, and persistence. This book is where you'll gain the knowledge.

As you may have gathered from the title, this volume is the second in a series. In *A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1*, I covered the foundational concepts. If you've read that volume or already know the concepts therein, you already know a lot! But now it's time to level up. Not only will you learn a wide variety of new

algorithms and data structures, but you'll also become more sophisticated in algorithmic analysis and design.

There are already books written on these subjects. However, if you've read any of them, you may have encountered the same problem I did: they're hard to understand! It's not just you—I can find myself reading and rereading the same paragraph in such a book many times before I get an inkling of what's going on. Many a developer has given up trying to learn these concepts, feeling incapable of grasping such complex ideas.

But here's the thing. Yes, these concepts are complex. But every complex concept is made up of a combination of *simple* concepts. It's not beyond anyone to master data structures and algorithms. The subject just needs to be *taught* correctly.

That's the entire point of this book. I've pulled every thread from the tapestry of each complex concept and laid them all out for you. By presenting each thread individually and in the correct sequence, I'll teach you these ideas so that you'll grasp them easily and clearly. Critically, there are literally hundreds of diagrams, all clear and beautiful, which will help clarify each concept for the visual learner. Finally, you'll have a lot of fun along the way. I write in an informal style and crack jokes whenever my editor lets me. Sometimes, the jokes are even funny.

Who Is This Book For?

You may be a professional software engineer of any experience level, a budding computer science student, or a code hobbyist. No matter what box you fit into, if you already know the basics of data structures and algorithms and want to level up, this is the book you're looking for.

If you've already read Volume 1 of this book, you're ready to dive into this book. (And, welcome back!)

If you *haven't* read Volume 1, I won't take it personally. However, you *do* need to be pretty familiar with certain topics to comprehend this volume. The topics, in alphabetical order, are these:

- Arrays
- Big O notation for time and space complexity
- Binary search
- Binary search trees
- Hash tables
- Heaps
- Recursion
- Sorting algorithms (the gist of Selection Sort and Quicksort)

In any case, I make many direct references to Volume 1 throughout this volume, so you can always gain more context if you need to.

What's in This Book?

In this volume, I don't simply cover a laundry list of additional data structures and algorithms. Yes, I cover those too, but the main focus of this book is to help you become more proficient in the analysis and design of algorithms.

Specifically, you'll find that there are three main themes that are threaded throughout this volume:

1. *Going beyond Big O.* While Big O notation is a useful and even crucial tool for algorithmic analysis, it has some significant limitations when applied to the real world. I'll show you where Big O notation falls short and how to use benchmarking and other forms of analysis to ensure that your code will truly be efficient in real life.

2. *Randomization.* There are many components you can integrate into an algorithm you're designing. One of the most useful but perhaps surprising of these components is randomness. We begin with the basics of randomization algorithms and then see how they can make your code more efficient in a wide variety of scenarios.

3. *Hardware.* An all-too-often overlooked factor in algorithm design is how your computer's hardware setup can impact the efficiency of your code. Sure, in an academic vacuum, how much memory your computer has shouldn't affect the Big O classification of an algorithm. However, in truth, a computer's hardware can have a *significant* impact on how fast your code will perform when you run it.

With regard to specific data structures and algorithms, here's a list of some of them that you'll encounter. I've listed them in the order in which they're presented in the book:

- Mergesort
- Fisher-Yates Shuffle
- Load balancing with the power of two choices
- LRU caches
- Red-black trees
- Randomized treaps
- External-memory algorithms
- B-trees
- Merging K sorted lists
- M/B-Way Mergesort
- Monte Carlo algorithms
- Random sampling
- Fermat's Primality Test
- Randomized hashing
- Hash function families
- Rabin-Karp substring search
- Sliding-window technique
- Boolean arrays
- Bit vectors
- Bit manipulation
- Bloom filters

In addition, each chapter contains exercises that will help you practice everything you've learned. Solutions to each exercise can be found at the back of the book.

A handful of exercises are special. You'll see that I marked some as being a "Puzzle" or "Exploration" or the like. You aren't expected to know the answer even if you've mastered the chapter. Instead, they ask you to apply ingenuity and see if you can stretch yourself even further. Occasionally, I'll also mark an exercise as a "New Concept" if I'll be teaching a brand-new idea in the associated solution.

How to Read This Book

In short, this book was designed to be read in order.

This isn't a book containing a hodgepodge of topics. I've painstakingly built up the concepts so that each chapter builds upon the previous one, and there's a kind of storyline arc that carries you through the book.

You *might* be able to skip to a chapter of interest, but you run the risk of not catching every bit of nuance since I'll have assumed that you read the previous chapters. Of course, after you've read the book once, you should be able to reread any chapter you wish for reference.

The exceptions are Chapters 7 and 8, which serve as a bit of a side quest. You could technically skip those and come back to them later if you'd like.

A Note About the Code

I strived to follow PEP 8 standards (for the most part) and write the code in such a way that it runs on Python Version 3.

That being said, I want to emphasize that the concepts in this book apply to virtually all coding languages, and I expect that some people who are not as familiar with Python will be reading this book. Because of this, I've sometimes avoided certain Python idioms that I thought would utterly confuse people coming from other languages. It's a tricky balance to keep the code Pythonic while also being welcoming to non-Python coders, but I hope that I've maintained an equilibrium that satisfies most readers.

Virtually all the code in this book can be downloaded online from the book's web page. This code repository contains automated tests, too! I encourage you to check out the tests, as they are not included in the actual book and can help clarify how to run the main modules.

You'll find some longer code snippets under the headings that read "Code Implementation." I certainly encourage you to study these code samples, but you don't necessarily need to understand every line to proceed to the next section of the book. If these long pieces of code are bogging you down, just skim (or skip) them for now.

Finally, it's important to note that the code in this book is not production-ready. My greatest focus has been to clarify the concept at hand, and while I did also try to make the code generally complete, I have not accounted for every edge case and optimization. There's certainly room for you to optimize the code further—so feel free to go crazy with that.

Online Resources

You can find more information about the book, download the source code for the code examples, and help improve the book by reporting errata, typos, and content suggestions on the book's web page.[1]

Additionally, I post updates about my writing at my website.[2] There, you can find more information about my books as well as video tutorials made by my colleagues and me in which we use the "common-sense" approach to explain all sorts of technologies and concepts. In particular, you might enjoy my Jay vs. Leetcode[3] video series, where I solve programming puzzles using many of the techniques discussed in this book.

Connecting

I enjoy connecting with my readers and invite you to find me on LinkedIn. [4] I'd gladly accept your connection request—just send a message that you're a reader of this book. I look forward to hearing from you!

Footnotes

[1] https://pragprog.com/titles/jwpython2

[2] https://commonsensedev.com

[3] https://www.commonsensedev.com/jay-vs-leetcode

[4] https://www.linkedin.com/in/jaywengrow

Chapter 1

Getting Things in Order with Mergesort

One of the most fundamental concepts central to understanding data structures and algorithms is understanding time complexity. *A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1* discussed this at great length, placing heavy emphasis on the use of Big O notation as a tool for articulating the speed of algorithms. Volume 2 takes things to the next level and adds more nuance to the conversation of time complexity. You'll learn that while counting an algorithm's steps and Big O serve as an important model for measuring time complexity, that's not the full story. This is because there are additional factors that can affect an algorithm's "true" speed. In this chapter, we'll take a look at one of those factors.

We'll also contrast two of the most famous "fast" sorting algorithms—Quicksort and Mergesort—and use the contrast to tease out the limits of the Big O model. Quicksort was covered in Volume 1, Chapter 13, and Mergesort is the main focus of this chapter. We'll look at basic array merges and then discover how they form the backbone of Mergesort. From there, we'll have an important conversation about algorithmic trade-offs. Finally, you'll discover how the counting-steps model is not the end-all of determining an algorithm's true speed.

Ready? Let's dive in.

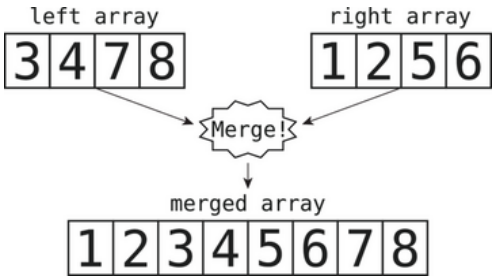
Merging Arrays

Mergesort sorts arrays by relying on another more basic algorithm known as *merging arrays*, or simply *merging*, for short. In this context, to merge arrays means to take two arrays that are already sorted, copy all of their values into a third array, and end up with the third array also completely sorted. Let's look at a basic example.

Let's say we have the arrays `[3, 4, 7, 8]` and `[1, 2, 5, 6]`. Note that each array is already sorted. I can't emphasize this enough: merging arrays only works if the arrays are *already sorted*.

The goal of merging is to take all of the values from both arrays and copy them into a third array, which will contain *all* the values in sorted order. The result of merging these two arrays will be: `[1, 2, 3, 4, 5, 6, 7, 8]`.

The following diagram shows the finished product of a merge:



The merging algorithm follows these steps:

1. Initialize a "left pointer" and have it point to the first index of the *left* array.

2. Initialize a "right pointer" and have it point at the first index of the *right* array.

3. Create a third, empty array. This will be the "merged" array. By the end, the merged array will contain all the values from the left and right

arrays in sorted order.

4. Run a loop until either the left pointer or the right pointer reaches the end of its array. Within the loop, do the following:

    1. Compare the value of the left pointer with the value of the right pointer and determine which value is *lower*;

    2. Take the lower value and append it to the merged array;

    3. Whichever pointer was pointing to the lower value gets incremented so that it points to the next index of its array. (If at any point the two values we're comparing are equal, we can arbitrarily append the value of the *left* array and move its pointer along.)

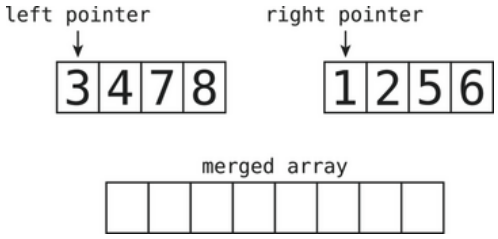
5. Once the loop is complete, either the left or right array will be "unfinished" in that it will still have values that were not yet copied to the merged array. This triggers the "final phase" of the algorithm.

6. Final phase: take all the remaining values of the "unfinished" array and append them, in order, to the merged array.

Let's now take a look at the merge algorithm in action, using an example.

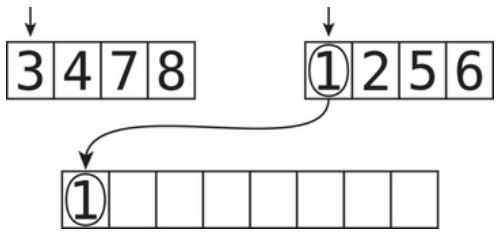
Merging in Action

The following diagram represents the arrays we want to merge. Note that the left and right pointers point to the beginning of each of their respective arrays. The merged array starts out as empty:

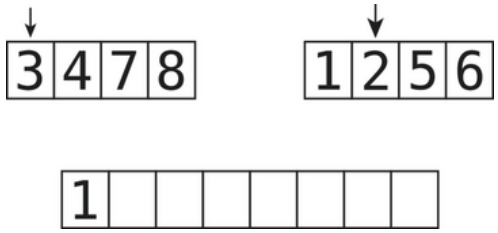


Now, let's walk through the process of merging them. Each of our "steps" will consist of two parts. Part A appends a value to the merged array, while Part B consists of moving either the left or right pointer.

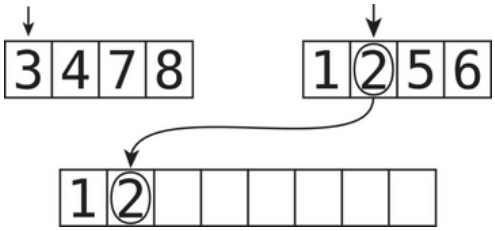
Step 1A: First, we compare the left pointer's value to the right pointer's value. We take whichever value is lower and append it to the array. In this case, the 1 is lower, so we append the 1:



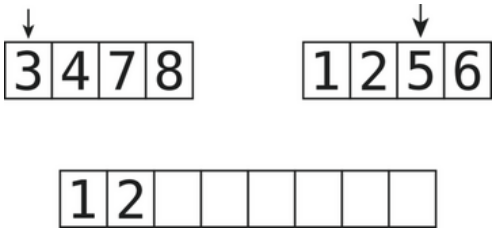
Step 1B: Because we appended a value of the right pointer, we now increment the right pointer so that it points to the next value of the right array:



Step 2A: We compare the left pointer's 3 with the right pointer's 2. Because 2 is lower than 3, we append the 2 to the merged array:

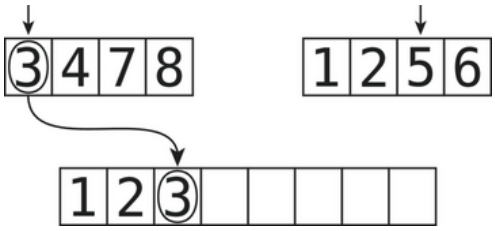


Step 2B: Since we once again appended a value belonging to the right pointer, we move the right pointer another notch rightward:



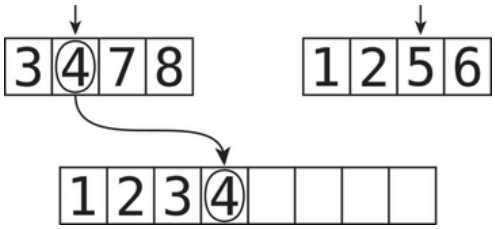
To expedite the remainder of this walkthrough, I'm only going to show visuals for the appends, that is, Part A of each step. I'll still mention the pointer movements (Part B), but won't show them in a dedicated diagram.

Step 3: We compare the values of the two pointers. The 3 is lower, so we append it to the merged array:



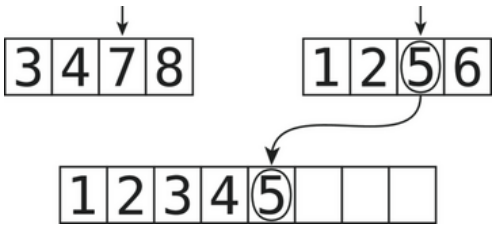
Accordingly, we'll move the left pointer along.

Step 4: We compare the 4 with the 5. We append the 4 to the merged array because it's lower:



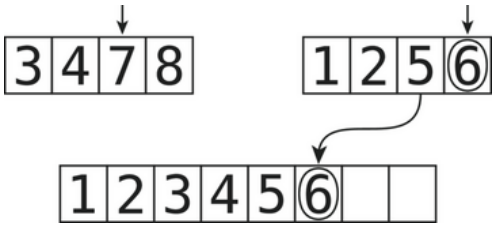
This means we'll also increment the left pointer again.

Step 5: We now compare the 7 with the 5. The right pointer's 5 is lower, so that's the value we add to the merged array:



At this point, we'll move the right pointer one notch rightward.

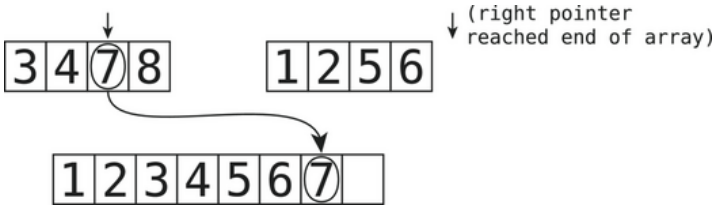
Step 6: Next, we compare the 7 with the 6. We copy the 6 to the merged array:



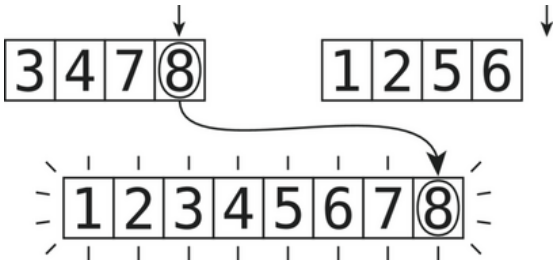
The 6 came from the right pointer, so we move that pointer another notch to the right.

Step 7: This is a noteworthy step because the right pointer has now moved beyond the end of the right array. This means we no longer need to perform any comparisons since we've exhausted all the values from the right array.

This triggers the final phase of the merge algorithm, in which we simply take all the values of the remaining array (in this case, the left array) and append each one to the merged array. So let's go ahead and append the 7 to the merged array:



Step 8: Similarly, we append the 8 to the merged array:



The merge algorithm is now complete! The merged array contains all the values from the left and right arrays, *and* is also sorted.

The Efficiency of Merging

Let's analyze how many steps merging takes. If we consider N to be the total number of values of both the left and right arrays combined, then there are at most about 2N steps. Here's why.

In the previous example, N is 8 because between the left and right arrays, there was a combined total of 8 values. Now, we take N steps to copy each of the N values into the merged array. In addition to the copy steps, we also perform comparison steps when we compare the values at the left and right pointers. In the previous example, we ended up making 6 comparisons. (The final two values of the left array didn't require a comparison since no comparisons take place during the final phase.) In a worst-case scenario, though, we'd have to make a comparison for all N values save for the last one.

When we add up our N copies and N-1 comparisons, we end up with 2N-1 steps. In terms of Big O notation, this reduces to O(N). As algorithms go, merging is very fast.

Now, here's a little spoiler regarding Mergesort, the algorithm we're leading up to. Mergesort itself is considered a very fast sorting algorithm. The reason is that the primary operation that Mergesort performs is merging, and merging itself is a super-fast algorithm, as you've seen.

Returning to our analysis of merging, it's worth noting that merging takes O(N) space. This is because we created a brand-new array and copied all the values into it. We'll look at the ramifications of this in a little bit.

Merging a Single Array

Imagine you want to sort the following array: `[3, 4, 7, 8, 1, 2, 5, 6]`. Are there any shortcuts we can take to sort this array?

While contemplating this, do you also notice anything familiar about this array?

If you look closely, you'll see that while the array as a whole is unsorted, *each half* of the array on its own is sorted. In fact, this array contains the *same values* as our prior merging example. We can take incredible advantage of this unique situation to sort the entire array in *just O(N) time*.

To accomplish this, all we need to do is treat each half of the array as if they were *separate* arrays, and merge them together! And this will take O(N) time since, as we've seen, merging has a time complexity of O(N).

I call this process "single-array merging" (not an official term), and it is the guts of Mergesort, as you'll soon see. I won't demonstrate single-array merging since it's essentially the same process we've already walked through.

I refer to this as single-array merging because typically, merging involves *two* arrays. Here, however, we start with a single array. The merge happens when we split this array into halves and then merge all the data back together again. As we've seen, this effectively sorts the array. And again, this only works if the two individual halves already happen to be sorted beforehand.

Tweaking Single-Array Merging

You've now seen that single-array merging is a fast way to sort a specific kind of array, namely, one whose two halves are already sorted.

However, we're now going to make a subtle tweak to single-array merging. In truth, we don't have to make this tweak, but we'll do so since the computer science literature does so, for a reason that I'll explain.

Generally, any given algorithm can come in a number of variants, and Mergesort is no exception. Each algorithm typically has a "classic" version

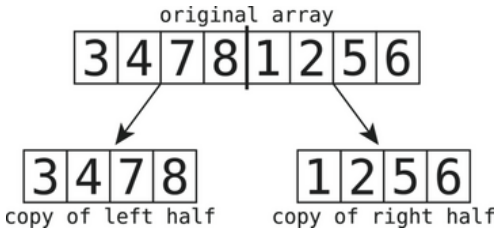
and a number of variants that optimize the algorithm in different ways.

One way to divide a sorting algorithm into two variants is as follows: a sorting algorithm can either sort the *original* input array, or it can produce a *brand-new* array that contains all the data in sorted order. Usually, the "classic" version of a sorting algorithm does the former; it sorts the original input array itself.

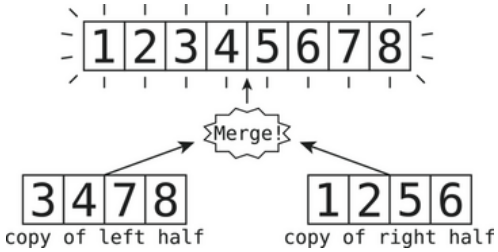
In this chapter, our focus will be on "classic" Mergesort, which sorts the *original* array. Because of this, classic Mergesort makes the following tweak to single-array merging.

Instead of dividing the single array and merging the two halves into a *brand-new* merged array, we're going to merge the data so that the original array *itself* gets sorted. As you'll see momentarily, we'll accomplish this by first copying the data elsewhere and then merging the data back into the original array.

First, we make *copies* of each half of the single array, creating two brand-new smaller subarrays:



Then, we merge these two "copies" and put the merged data *back into the original array* by overwriting all the original values:



In other words, our first version of single-array merging didn't copy any data before performing the actual merge. We simply merged the two halves of the array into a brand-new array. With this new tweak, however, we first make copies of the two halves and only then merge the data. And in this tweaked version, when we do merge the data, we merge it back into the original array, overwriting its old values.

It's striking that we have to perform an extra N steps for this "tweaked" merge. That is, in our original version of merging, we spent N steps appending the merged data to a brand-new merged array. However, with our tweak, we first spend N steps copying the data and then *another* N steps overwriting the original array with the merged data.

So, because our original version takes, at most, 2N-1 steps, this tweaked version takes *3N*-1 steps. Luckily, this is still considered fast and is also considered O(N).

Again, the reason for this tweak is somewhat arbitrary; we did it to look at Mergesort in its most classical form before moving on to any of its variants. As I've said, the reason why the classical Mergesort algorithm does this is because it can thereby sort the *original* array, which is what classical sorting algorithms tend to do. The variant of Mergesort that uses our original version of merging, on the other hand, doesn't sort the original array but instead produces a *brand-new* array that's sorted.

Code Implementation: Single-Array Merging

In any case, we're going to run with this tweaked version of merging. Before we look at all of the code, let's first focus on the method signature:

```
def merge(copy_of_left_half, copy_of_right_half, original_array):
```

We call `merge` on a single array like the one we had before, namely, `[3, 4, 7, 8, 1, 2, 5, 6]`. That is, this method is designed to work on an array that has two sorted halves, even though the array as a whole is unsorted. If you're

wondering why such a method is useful, given that such an array is pretty rare, the answer will become clear when I reveal the entire Mergesort algorithm.

To use our method, we'll pass in the original array, which is the method's third argument. But, we'll also first make a copy of each half of the array, and pass those in as the `copy_of_left_half` and `copy_of_right_half` arguments. Here's an example of how we'll call this method:

```
array = [0, 2, 5, 6, -1, 6, 7, 9]
midpoint = len(array) // 2
copy_of_left_half = array[:midpoint]
copy_of_right_half = array[midpoint:]
mergesort.merge(copy_of_left_half, copy_of_right_half, array)
```

It might seem strange that we have to make copies of the left and right halves before calling the `merge` method. After all, can't the `merge` method do that itself? Once it receives the `original_array`, it should be able to make copies of the two halves. Again, the answer to this will become clear when we reveal the entire Mergesort algorithm. (Don't worry—we'll get there soon!)

Now, let's get to the meat of the `merge` method. Here's the complete code:

```
def merge(copy_of_left_half, copy_of_right_half, original_array):
    left_pointer = 0
    right_pointer = 0
    array_pointer = 0

    while left_pointer < len(copy_of_left_half) \
      and right_pointer < len(copy_of_right_half):
        if copy_of_left_half[left_pointer] <=
copy_of_right_half[right_pointer]:
            original_array[array_pointer] = copy_of_left_half[left_pointer]
            left_pointer += 1
        else:   # the value at right pointer is greater than the left
pointer
            original_array[array_pointer] =
copy_of_right_half[right_pointer]
```

```
            right_pointer += 1

        array_pointer += 1

    # Append any remaining elements from the left half
    if left_pointer < len(copy_of_left_half):
        original_array[array_pointer:] = copy_of_left_half[left_pointer:]
    # Append any remaining elements from the right half
    if right_pointer < len(copy_of_right_half):
        original_array[array_pointer:] = copy_of_right_half[right_pointer:]
```

Let's walk through this code one piece at a time.

First, we set up our left and right pointers to start at index `0`. We also initialize an `array_pointer` which points to index `0` of the `original_array`. We'll need this since we're going to be overwriting the values of the `original_array`, and this pointer will point to the index that we're going to overwrite.

The next bit of code then begins a loop that lasts as long as neither the `left_pointer` nor the `right_pointer` has reached the end of their respective arrays.

```
while left_pointer < len(copy_of_left_half) \
  and right_pointer < len(copy_of_right_half):
```

So, as soon as either pointer reaches the end of its array, this loop will terminate.

We then compare the value at the `left_pointer` with the value at the `right_pointer`. If the `left_pointer`'s value is the lower one (or the two values are equal), the `left_pointer`'s value gets copied to the `original_array` at whatever index the `array_pointer` is at. We also increment the `left_pointer` by 1:

```
if copy_of_left_half[left_pointer] <= copy_of_right_half[right_pointer]:
    original_array[array_pointer] = copy_of_left_half[left_pointer]
    left_pointer += 1
```

Note that this overwrites a value from the `original_array`.

If, on the other hand, the `right_pointer`'s value is lower, we copy it to the `original_array` and increment the `right_pointer`:

```
    else:
        original_array[array_pointer] = copy_of_right_half[right_pointer]
        right_pointer += 1
```

In either case, we then increment the `array_pointer` so it's ready to overwrite the `original_array`'s next value:

```
    array_pointer += 1
```

Once the loop is done, either the `copy_of_left_half` or `copy_of_right_half` has at least one value in it that we didn't yet process. So, we begin the final phase. In the following code, we append the remaining values from the "unfinished" array:

```
    if left_pointer < len(copy_of_left_half):
        original_array[array_pointer:] = copy_of_left_half[left_pointer:]

    if right_pointer < len(copy_of_right_half):
        original_array[array_pointer:] = copy_of_right_half[right_pointer:]
```

Note that only one of these two conditional statements will be triggered since only one pointer will be past its array.

And that's all there is to single-array merging!

In a vacuum, sorting a single array using merging seems to be pretty moot. After all, it only works in a *very* specific case where an array happens to have its two halves perfectly sorted. How often would we encounter such an array? And how would our program even know that it's dealing with such an array? If only we could somehow ensure that the array we're sorting is such an array.

Well, reader, we can.

Mergesort

With Mergesort, you can make sure that your array contains two sorted halves before it performs a single-array merge, thereby sorting the array.

First, we'll look at the steps of the Mergesort algorithm, and then we'll walk through an example. The recursion may seem intimidating at first, but I assure you that the diagrams will clarify everything.

Let's do it!

Here are the steps:

1. Create two new arrays. One is a copy of the array's left half, and the other is a copy of the array's right half.

2. Recursively perform Mergesort on the left copy. (Indeed, we're already in the middle of Mergesort right now; that's recursion for you.) The base case is when the array we perform Mergesort on contains one element or fewer.

3. Recursively perform Mergesort on the right copy. (The base case is the same as the previous step.)

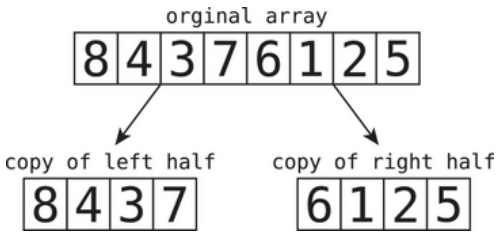
4. Merge the left and right copies back into the original array.

That's pretty much it! Let's now visualize this algorithm.

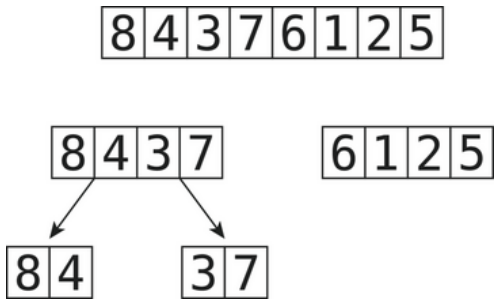
Mergesort in Action

Assume that we want to sort the array `[8, 4, 3, 7, 6, 1, 2, 5]`. Note that this is *not* the type of array whose two halves are already sorted. In fact, it's a complete mess.

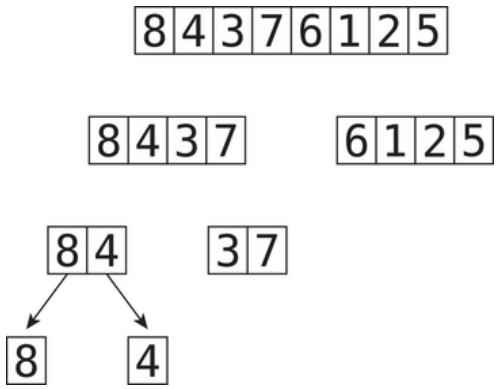
Step 1: First, we make copies of the two halves of the array:



Step 2: We recursively perform Mergesort on the left copy. In code, this would be written as `mergesort([8, 4, 3, 7])`. This, in turn, will make copies of the two halves of `[8, 4, 3, 7]`:

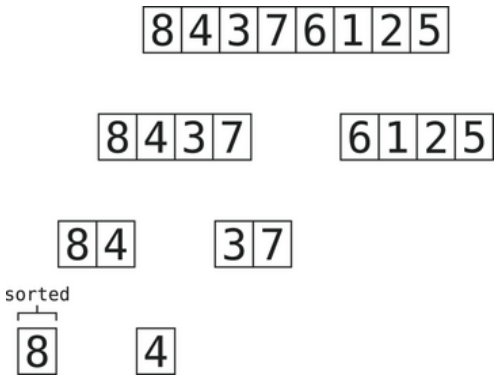


Step 3: We recursively perform Mergesort on the left copy (which I'll call left *half* going forward). This means we call `mergesort([8, 4])`, and thereby create two new subarrays, `[8]` and `[4]`:

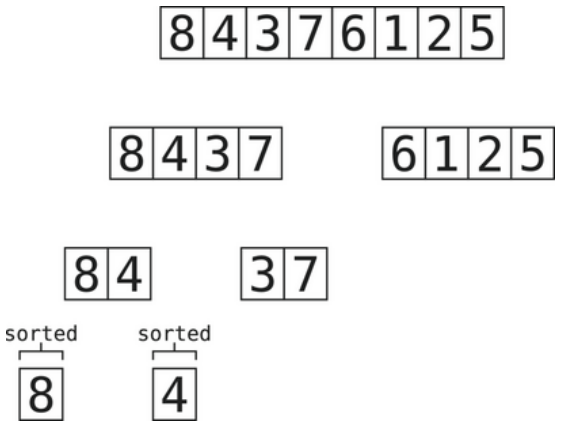


Step 4: We recursively Mergesort (yes, I'm making it a verb now) the `[8]`. Since it's the base case of an array of size 1, we don't do anything else within this call, and `mergesort([8])` is completed.

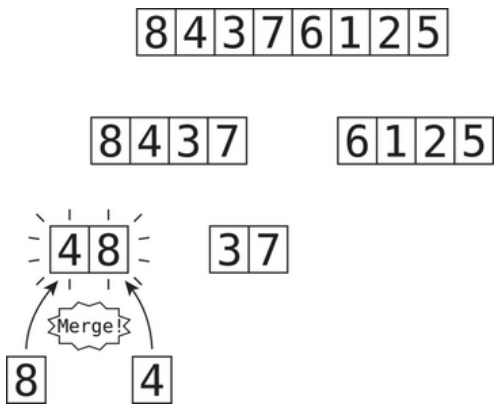
The reason why a single-element array is the base case is because *a single element is, by definition, sorted*! (After all, it's certainly not unsorted, right?) As such, we mark the `[8]` as sorted:



Step 5: We now Mergesort the right half of `[8, 4]` by calling `mergesort([4])`. The `[4]`, too, is an array of size 1, so it is considered sorted:

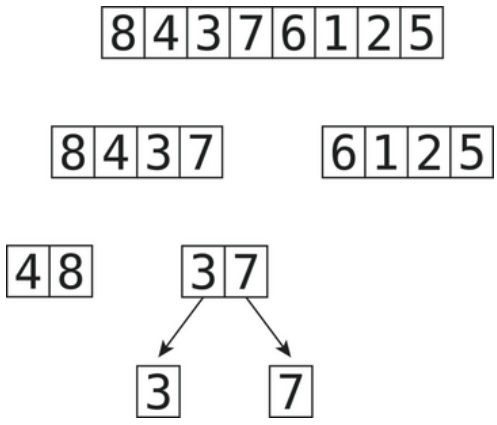


Step 6: With `mergesort([8])` and `mergesort([4])` complete, we now proceed to the next step of the `mergesort([8, 4])` call, which is to merge the two halves:



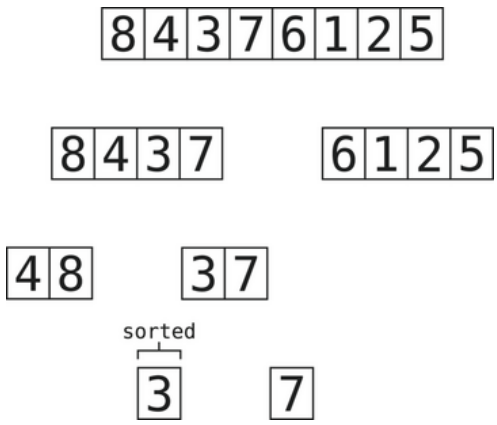
Note that the `4` and `8` are now sorted relative to each other.

Step 7: We've completed `mergesort([8, 4])`, which brings us back to call `mergesort([8, 4, 3, 7])`. We've already Mergesorted the left half, so now we Mergesort the right half and call `mergesort([3, 7])`:

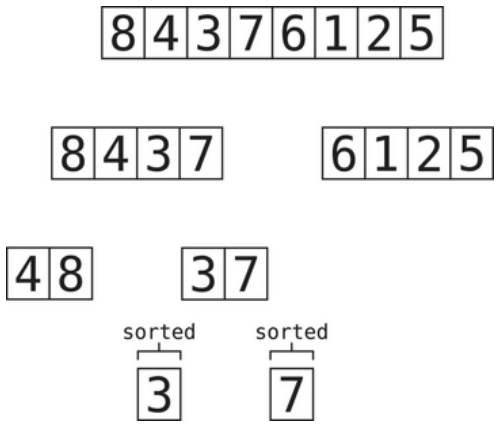


Now, you and I both know that `[3, 7]` is already sorted, but the computer doesn't know that yet since it doesn't have eyeballs.

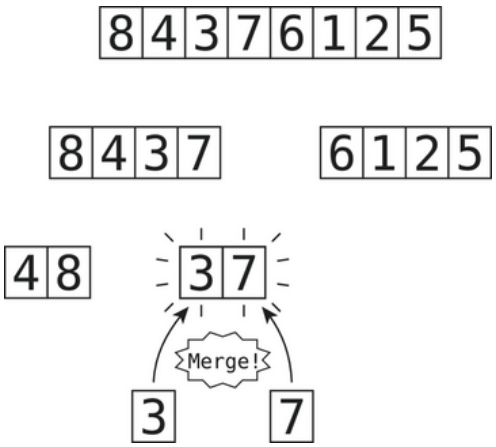
Step 8: We Mergesort the left half of `[3, 7]` by calling `mergesort([3])`. This is a base case, so the `[3]` is now sorted:



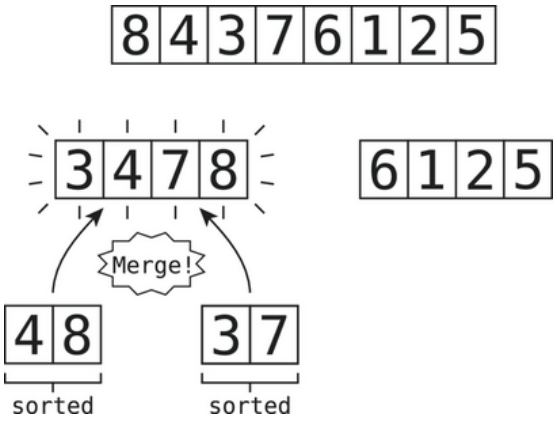
Step 9: We then Mergesort the right half of `[3, 7]` by calling `mergesort([7])`. This, too, is a base case, so `mergesort([7])` is complete:



Step 10: Now that `mergesort([3, 7])` has Mergesorted both its left and right halves, we now merge the halves together:

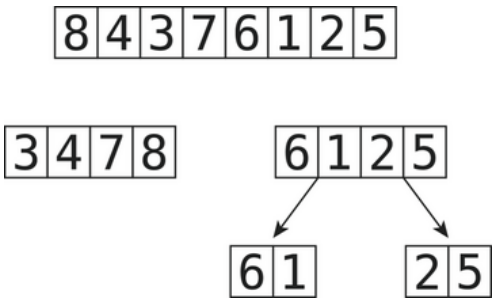


Step 11: This is where things get exciting. We're back within the call of `mergesort([8, 4, 3, 7])`. We've already Mergesorted its left half. And we've already Mergesorted its right half. This means we get to merge the two halves together. Boom!

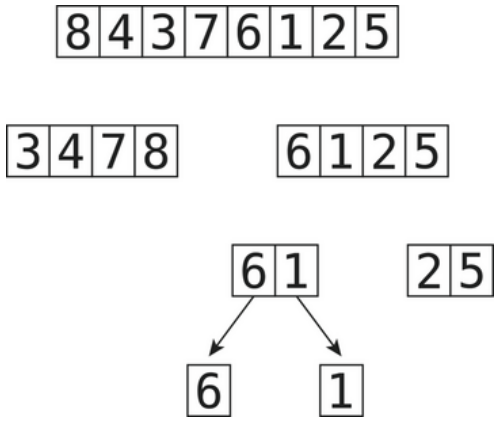


The `[3, 4, 7, 8]` is now sorted.

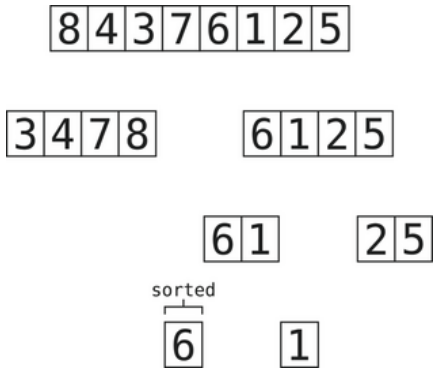
Step 12: We've completed Mergesorting the left half of our original array `[8, 4, 3, 7, 6, 1, 2, 5]`. Now, it's time to Mergesort the right half, `[6, 1, 2, 5]`:



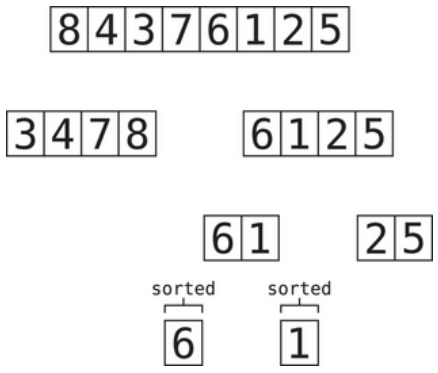
Step 13: We Mergesort the left half of `[6, 1, 2, 5]` by calling `mergesort([6, 1])`:



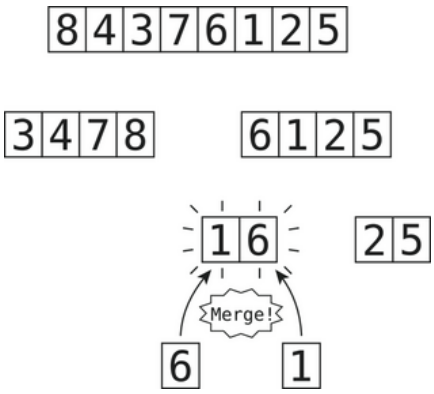
Step 14: We Mergesort the left half of `[6, 1]`, calling `mergesort([6])`. This is a base case, so the call ends abruptly:



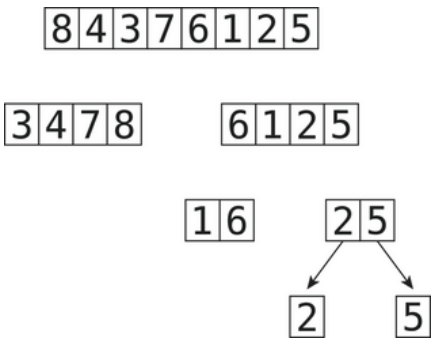
Step 15: We Mergesort the right half of `[6, 1]` by calling `mergesort([1])`:



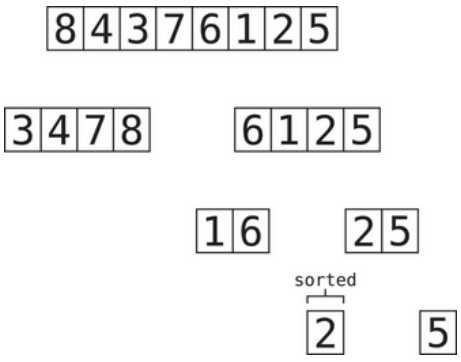
Step 16: We're done sorting the two halves of `[6, 1]`, so we now merge them:



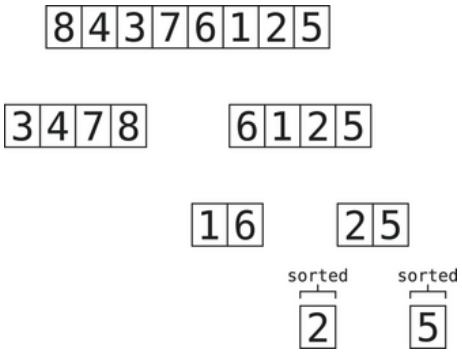
Step 17: We're back to `mergesort([6, 1, 2, 5])`. Since we completed Mergesorting its left half, we now proceed to Mergesort its right half with `mergesort([2, 5])`:



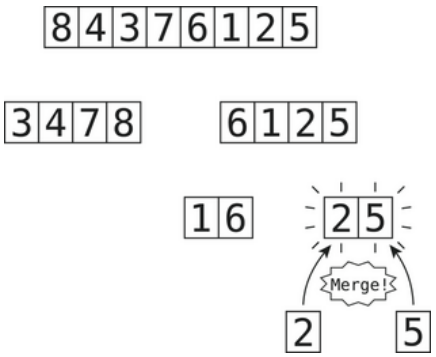
Step 18: We Mergesort `[2]`:



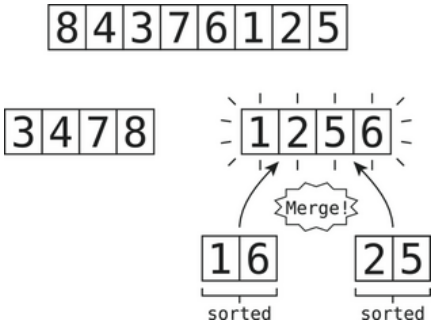
Step 19: We Mergesort `[5]`:



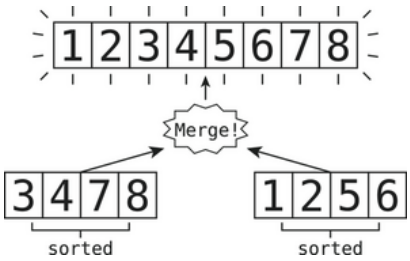
Step 20: We merge the two halves together:



Step 21: We're back at `mergesort([6, 1, 2, 5])`. We've Mergesorted both halves, so we now merge the two halves together. Boom!



Step 22: Okay, we're at the climax now. We're back at our original call of `mergesort([8, 4, 3, 7, 6, 1, 2, 5])`. We've Mergesorted both halves of the array. You know what this means, right?



BOOM! Our original array is now completely sorted.

Code Implementation: Mergesort

Here's the code for Mergesort. It's surprisingly concise:

```
def mergesort(array):
    if len(array) <= 1: return

    midpoint = len(array) // 2
    copy_of_left_half = array[:midpoint]
    copy_of_right_half = array[midpoint:]

    mergesort(copy_of_left_half)
    mergesort(copy_of_right_half)
    merge(copy_of_left_half, copy_of_right_half, array)
```

First, the call simply ends abruptly if the size of the `array` is 1 or lower. Again, this is the base case.

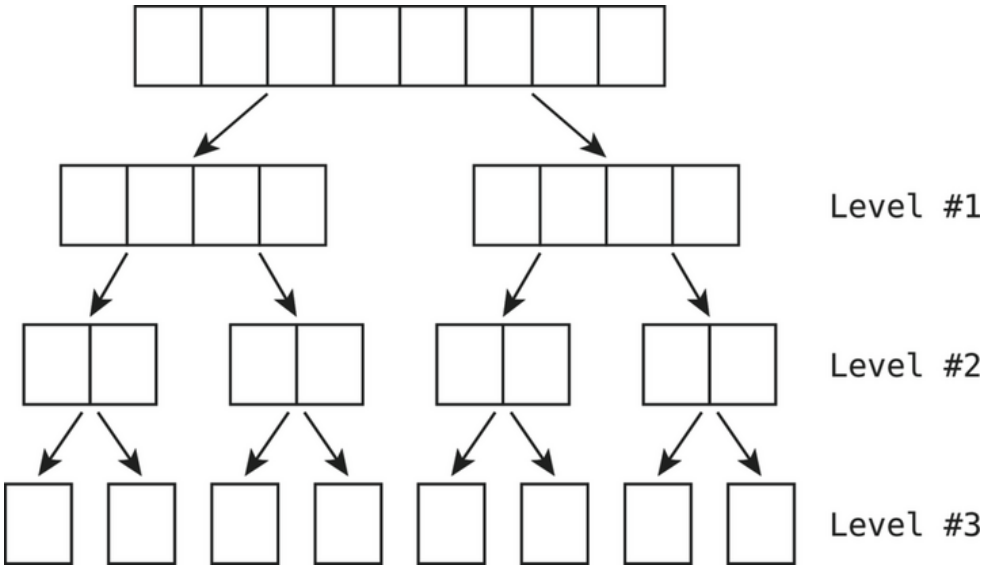
Next, we make copies of the left and right halves. After that, we Mergesort the left half, and after that, we Mergesort the right half. Finally, we merge the two halves using our `merge` function from earlier in this chapter.

And that's it!

The Efficiency of Mergesort

Let's figure out the efficiency of Mergesort. To help with this, we'll first look at Mergesort from a bird's-eye view, as this will put everything in perspective.

Here's a visual that shows the entire Mergesort process in one fell swoop. I removed all of the array data since it's not relevant to our analysis.

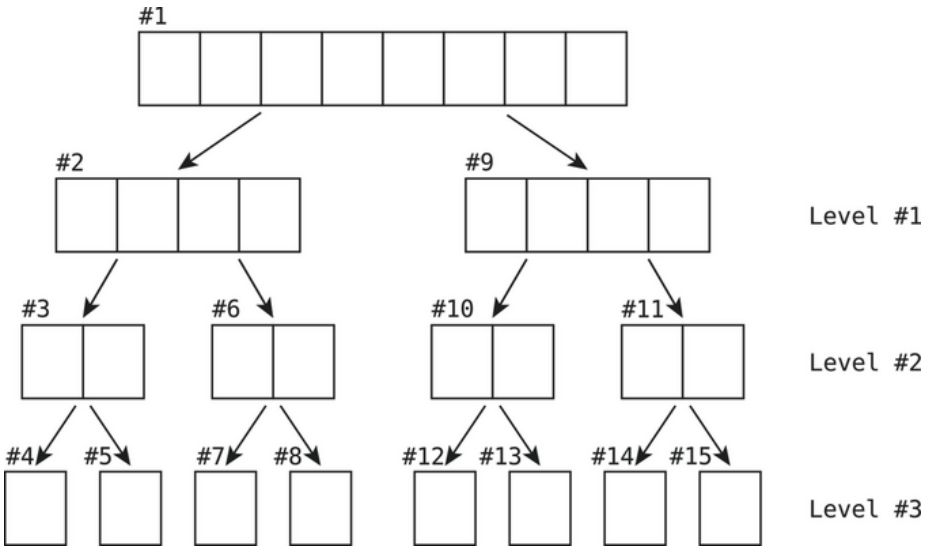


Looking at the diagram, notice that the original array is repeatedly broken down into halves until we're eventually left with single-element arrays. Note that there are 3 "levels"; that is, for this array of size 8, it takes 3 halvings until we break up the array into single-element subarrays.

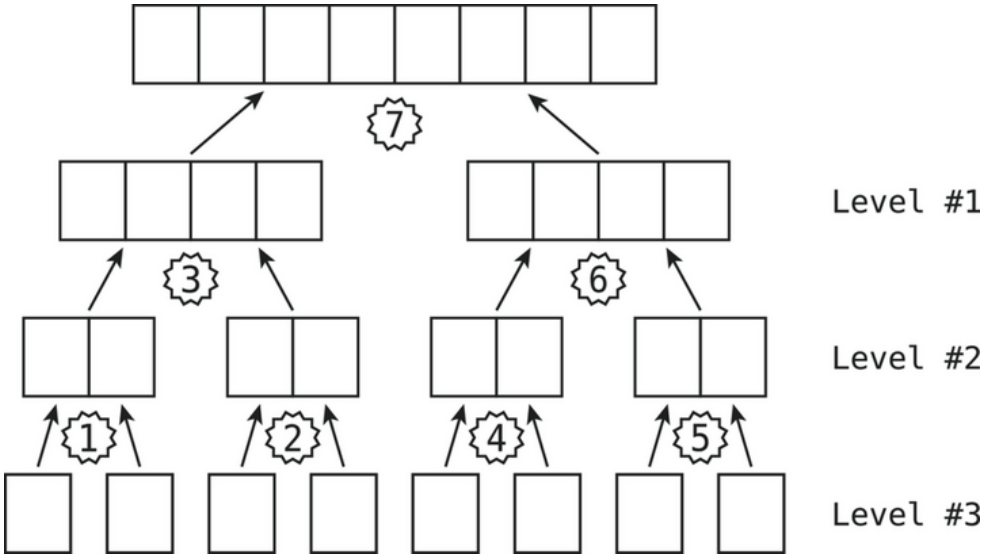
In more general terms, when you have an array that is size N, it takes *log N* halvings until the array is completely broken down into single-element subarrays. This may be more intuitive if you recall our unique definition of log N from Volume 1, Chapter 3. That is, log N is the number of times it takes to halve N until we arrive at 1. In our example, where N is 8, *log N* is 3, and that's why we end up with 3 levels in the diagram.

Next, let's take a look at the order in which the `mergesort` function is called on each of these subarrays.

There are 15 such calls:



Also of interest is the order in which the *merges* take place:



When we contrast the two previous diagrams, we see that the order of merges is different than the order of `mergesort` function calls.

All in all, there are 7 merges. Merges 1, 2, 4, and 5 all take place at Level #3. Merges 3 and 6 operate on all four subarrays at Level #2, and Merge 7

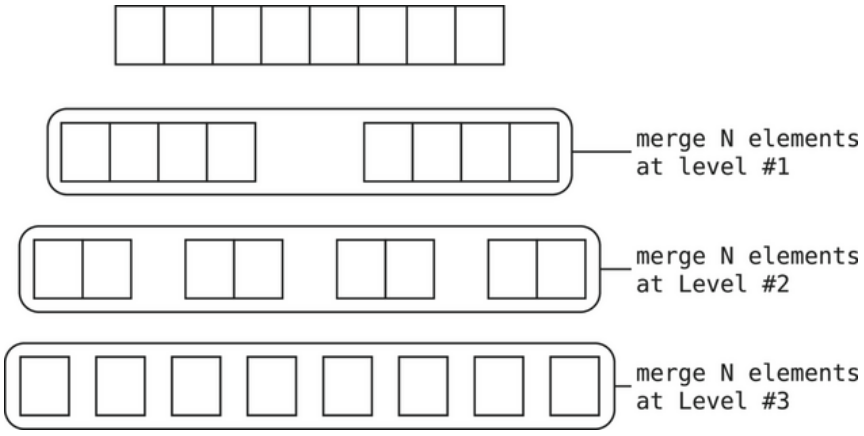
merges the two subarrays of Level #1.

The Time Complexity of Mergesort

Let's get down to brass tacks. How fast is Mergesort?

To break this down, let's analyze how many steps take place relative to the N elements of the original array that we're sorting. Using the "levels" concept we demonstrated in the previous visuals, we find that each level contains N elements, and that we perform merges on all N elements at each level.

The following diagram highlights this point:



So, in our example, where the original array was of size 8, we end up merging 8 elements 3 times. That is, there are 3 levels, and on each level, we merge 8 elements. While it's true that we perform a different number of merges at each level, it doesn't change the fact that the *number of elements* being merged remains the same at each level. That is, we may perform 4 merges at Level #3 and only 2 merges at Level #2, but at both levels, we merge 8 elements.

To generalize this in terms of N, we'd say that we perform merges on *N elements multiplied by the number of levels*. Given that for N elements there are log N levels, we can conclude that we merge a total of N log N elements since:

N elements * log N levels = N log N merged elements.

Now, Mergesort's primary operation is to perform merges; it doesn't do much else. And we've already established that merging N elements takes N steps. In other words, a merge takes one step per element being merged. Therefore, since Mergesort merges a total of N log N elements, this means that Mergesort takes N log N steps. In Big O, we'd say that Mergesort takes *O(N log N) time*.

In truth, I noted earlier that a single-array merge can take about 3N steps when we factor in the extra copying of data (plus the first copying and the comparisons). Accordingly, Mergesort may take 3N log N steps. However, this still reduces down to O(N log N).

Next up, let's analyze how much memory Mergesort consumes.

The Space Complexity of Mergesort

Mergesort's memory consumption occurs when the `mergesort` function makes copies of the array it's sorting:

```
copy_of_left_half = array[:midpoint]
copy_of_right_half = array[midpoint:]
```

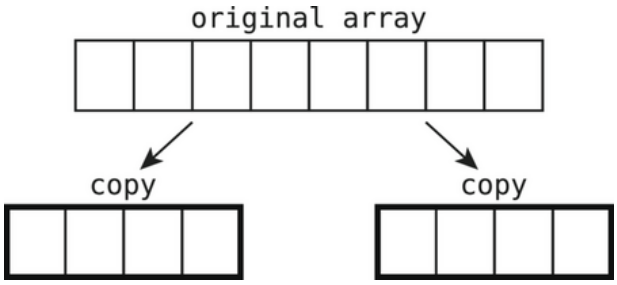
Again, the code makes a copy of the array's left and right halves, which in total takes up the same amount of space as the array itself. Essentially, in addition to the array itself, we have a copy of it as well. Furthermore, these copies aren't made just once. Each call of `mergesort` copies whatever array it's acting upon.

To figure out how much extra space Mergesort takes in total, recall that each call of Mergesort recursively calls itself on the left and right copies after creating them:

```
mergesort(copy_of_left_half)
mergesort(copy_of_right_half)
```

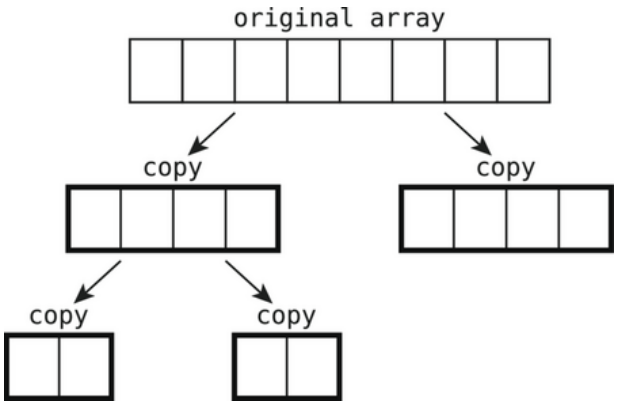
Let's walk through some of the Mergesort steps again and keep track of how many array copies are kept in memory at a given time.

When we first call `mergesort` on the original array, we create copies of the original array's left and right halves. The copies are highlighted in the following image:



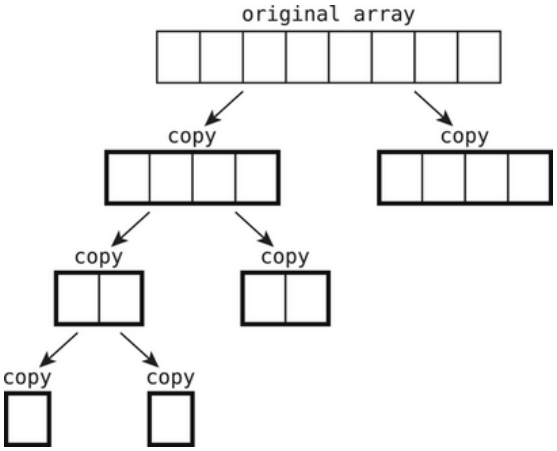
These copies store an extra N elements in memory, which, in our example, is an additional 8 elements.

We then call `mergesort` on the left copy, which makes copies of itself:



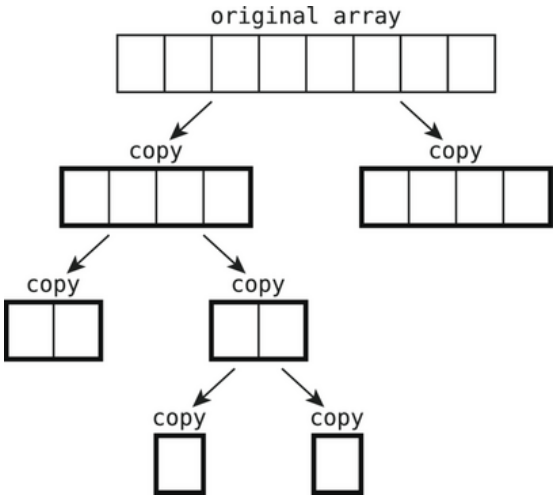
This means we have to store yet another N/2 elements. That is, N is 8, and we've created another 4 elements.

Then, we call `mergesort` yet again, this time on a copy of a copy:



We create another N/4 elements.

When we unwind the recursion call stack, some of these copies start to disappear, but then new ones are made. For example, if you jump ahead a few steps in the Mergesort algorithm, you'll see that we have the following set of copies:



So, in this example, we have at most N+(N/2)+(N/4) copied elements at a given time. If we try plugging in different examples of N, we'll find that this always comes out to be 2N-2 extra elements.

Here are a few examples.

In our example, N is 8, so 2N is 16, and we store an extra 14 elements. This is 2N-2.

If N is 16, we end up with 4 levels of halvings. At a given time, we'll have N+(N/2)+(N/4)+(N/8) extra elements, which totals 30 elements, which is also 2N-2 (since 2N is 32).

Similarly, if the original array contains 64 values, the total number of extra elements comes out to be 126. Again, this ends up being 2N-2.

Beyond the extra copies, recursion itself takes up additional space since the computer has to keep the call stack in memory. This was discussed in Volume 1, Chapter 19. In our case, this amounts to log N units of memory since we have to keep track of up to log N levels at most. Relative to 2N-2, or even N alone, log N is pretty negligible.

Rounding things off, it emerges that Mergesort takes about 2N extra units of space beyond the original array. When it comes to Big O notation, though, this reduces to O(N).

Comparing Mergesort and Quicksort: Lessons Learned

We now arrive at a pivotal question. Which sorting algorithm is better, Mergesort or Quicksort? Quicksort was the fastest sorting algorithm we covered in Volume 1, but let's look at how Mergesort measures up to it. (To follow this discussion, you don't need to remember all the nitty-gritty details of Quicksort; I'll remind you of the pertinent facts.)

As discussed in Volume 1, Quicksort has an average speed of O(N log N). Now, given that Mergesort also runs, on average, in O(N log N) time, it would seem at first glance that the two algorithms are equally fast. In reality, though, Quicksort is faster than Mergesort in what I call *actual time*.

I use the term *actual time* to refer to time as measured in minutes and seconds, and not Big O Notation. A major theme of this volume is that although time complexity can be measured in terms of Big O and counting steps, we shouldn't ignore the *classic* definition of time completely.

So, Quicksort and Mergesort have the same time complexity in terms of Big O. However, in actual time, the reality is that for the same N elements, Quicksort takes fewer seconds to run than Mergesort does.

Why is this so?

Not All Steps Are Created Equal

One *potential* explanation for why Quicksort is faster than Mergesort in actual time is that while Mergesort is O(N log N) in terms of Big O notation, we saw earlier that it takes about 3N log N steps. If we look back at our analysis of Quicksort in Volume 1, we find that Quicksort takes closer to N log N steps.

However, this answer isn't so simple. First, while our implementation of Mergesort earlier took 3N log N steps, there are other variants that take closer to 2N log N steps.

On the flip side, while Quicksort seems to take just N log N steps, that's only limited to a *best*-case scenario where the pivot value keeps ending up smack in the middle of the array. However, in more typical cases, computer scientists have found that Quicksort generally takes closer to 2N log N steps.

So, now we have a real puzzle. If the fastest variants of Mergesort and Quicksort both take around 2N log N steps in average scenarios, why is Quicksort faster? To make things even more puzzling, computer scientists report that the most efficient variants of Quicksort are from *two to three times faster* than the most efficient variants of Mergesort!

This brings us to our first major lesson, which is going to stir things up.

We see that even when two algorithms take *the same number of steps*, one algorithm can still be significantly faster than the other in actual time.

In other words, not only can two algorithms have the same Big O but different speeds in actual time, but even if the two algorithms have the same number of steps, one algorithm may still get the job done more quickly.

This can be true for a variety of reasons. One reason has something to do with a concept called *spatial locality*. We'll explore this idea further in Chapter 4.

However, let's focus on another particular reason for now.

What we consider a single "step" in a high-level language like Python may consist of *numerous* steps in the *lower-level processes* of the computer. A computer breaks high-level code down into low-level code, generally called

machine code, and one step in Python might involve *10* steps of machine code.

Because of this, different types of Python steps can have varying speeds. For example, it's possible that a Python step that *compares* two values will create three machine-code commands, while a step of *swapping* two values may involve *10* machine-code commands.

Exploring Bytecode

To give you a taste of what I'm talking about, let's take a look at how a simple Python step breaks down into multiple steps closer to the machine level.

In truth, Python doesn't translate *directly* into machine code. Rather, the Python code first gets converted to something called *bytecode*. Bytecode is code that is lower-level than Python, but higher-level than machine code. What's cool is that it's super easy to see the bytecode created by our Python code.

Let's create a file called `byte_code_example.py` and insert the following Python code:

```
x = 1
x += 3
```

It's pretty reasonable to say that this code represents *two* Python steps.

You can see the bytecode generated by the Python code by running the following command in your console:

```
python -m dis byte_code_example.py
```

This spits out bytecode, which for me looks something like this:

```
1           0 LOAD_CONST               0 (1)
            3 STORE_NAME               0 (x)
```

```
  2           6 LOAD_NAME                0 (x)
              9 LOAD_CONST               1 (3)
             12 INPLACE_ADD
             13 STORE_NAME               0 (x)
             16 LOAD_CONST               2 (None)
             19 RETURN_VALUE
```

Each line here represents a bytecode step. It shows that our two Python steps trigger *eight* bytecode steps. Intriguing! (It can be fun to go a little crazy and explore the bytecode of all the Python code we write.)

So, *this* is one plausible reason as to why Quicksort is faster than Mergesort despite the fact that they involve a similar number of steps. Quicksort might break down into fewer bytecode or machine-code steps than Mergesort does.

Note that even if Algorithm A translates to 10 bytecode steps and Algorithm B translates to 20 bytecode steps, this is no guarantee that Algorithm A will run faster than Algorithm B. It may give us a *hint* as to the speed of our code, but it will not give us definitive results. However, it's one factor to consider.

The Limits of Big O Notation

Well, isn't that a monkey wrench? We already knew that Big O notation had some limitations. For example, we know from Volume 1, Chapter 5, that two algorithms can be classified as O(N) even though Algorithm A takes N steps and Algorithm B takes 10N steps. But now you've learned that even when two algorithms take the same number of Python steps, one algorithm can still be significantly faster than the other!

This being the case, how are we supposed to ever truly know which algorithms are the most efficient? In truth, even if we were experts in machine code and spatial locality, it's still hard to predict the effects of these factors.

The surprising answer to these questions is that, indeed, Big O notation is limited! It's a framework that allows us to *approximate* relative speeds of competing algorithms, but it's far from foolproof. The closer that two algorithms are in terms of the number of steps, the harder it becomes to know which algorithm is faster in actual time.

But there's good news.

There are more precise tools than Big O out there. One such tool, called *benchmarking*, is the subject of the next chapter. If Big O is a butter knife, then benchmarking is a scalpel.

This isn't to say that we'll be throwing out Big O. Far from it! Big O notation still remains a fundamental tool for conceptualizing algorithm efficiency and grouping algorithms into different general categories. So, we'll still be using Big O throughout this book extensively. After all, you wouldn't use a scalpel to butter your bagel.

Trade-Offs

Let's get back to the issue at hand. Which algorithm is better: Quicksort or Mergesort?

We've seen that Quicksort is indeed faster than Mergesort in actual time.

Now, not only is Quicksort faster than Mergesort, but Quicksort also consumes *less space* than Mergesort. This is because Quicksort doesn't make any copies of the original array, and the only extra space consumed is the recursion call stack, which is all of O(log N). Mergesort, on the other hand, takes up O(N) extra space, as I explained earlier.

However, it could be argued that Mergesort has one advantage over Quicksort: Mergesort is faster than Quicksort in a *worst-case scenario*. (In Chapter 3, we're going to pull the rug out from under this advantage, but we'll run with it for now.)

So, here's the deal. In average scenarios, both Quicksort and Mergesort run in O(N log N) time. However, Quicksort has a potential Achilles' heel: there are cases where it slows down to $O(N^2)$ time. Specifically, if an array is already completely or mostly sorted, Quicksort will take about N^2 steps. You can look back at Volume 1, Chapter 13, where we discussed the reason for this, but for now, you can take my word for it.

On the other hand, Mergesort *always* runs in O(N log N) time. Essentially, there *is* no worst-case scenario for Mergesort, as all scenarios get processed at the same speed. If you walk through the steps of Mergesort for any array, you can see for yourself that it will always break down the array in log N levels and will always merge N elements at each level.

So, when choosing whether to use Quicksort or Mergesort, there are pros and cons on each side. On one hand, it may be worth using Quicksort because of its average-case efficiency, even though you may risk encountering a presorted array and having a slowdown. Alternatively, it may be worth it to choose Mergesort and *guarantee* that your speed is never slower than O(N log N), even though, on average, Mergesort will be slower and consume more memory.

Ultimately, there is no *right* choice. Only you can decide, based on your software requirements, which choice is better for you. And this is the idea of a *trade-off*.

We're certainly no strangers to trade-offs, as we've encountered this a number of times throughout Volume 1. One example is the trade-off between time and space discussed in Volume 1, Chapter 19. If Algorithm "A" is faster but consumes more memory, and Algorithm "B" is slower but consumes less memory, which do we choose? Again, it's a trade-off, and only you can decide which is best for your particular situation.

In truth, almost every technology decision involves a question of trade-offs, and this is certainly so when it comes to data structures and algorithms. I'm

placing a special emphasis on trade-offs here because trade-offs will be another major theme of this volume.

We're not done analyzing the trade-offs of Mergesort and Quicksort yet. We've certainly discussed some of the major factors at play, but there's more analysis to come in the following chapters.

Wrapping Up

You're only getting started, but you've already learned a few things. First, you learned how Mergesort works. Mergesort is one of the classic algorithms of computer science, so it's good to have that under your belt.

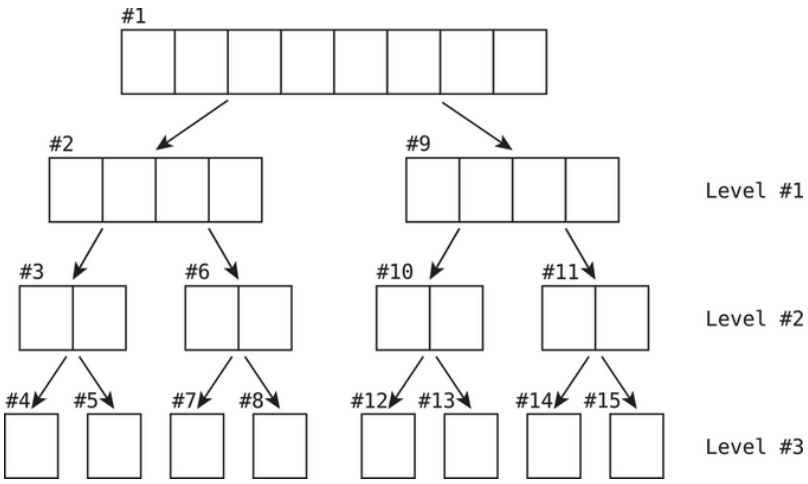
In addition, you learned there is no one perfect algorithm. Weighing two competing algorithms is about trade-offs. There are benefits and drawbacks to each algorithm, and it's up to you to decide which set of pros and cons will fit your application best. This will be a running theme throughout this book.

Finally, you discovered how two algorithms can have the same number of Python steps and yet have significantly different speeds in actual time. In the next chapter, you'll learn how to grapple with this reality.

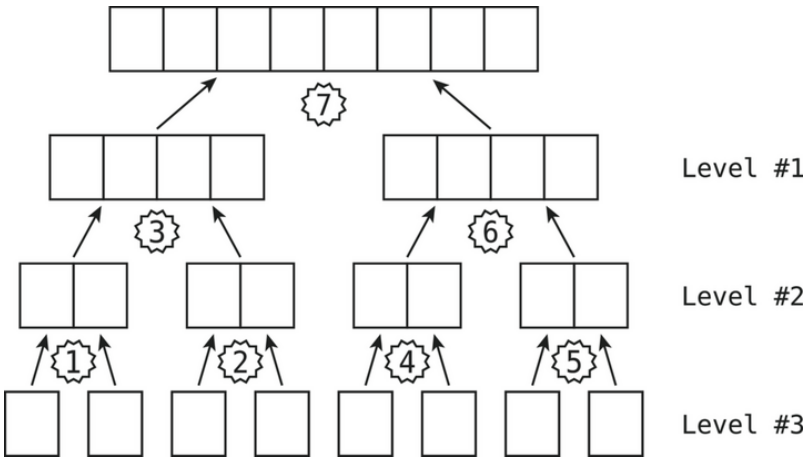
Exercises

The following exercises provide you with the opportunity to practice with Mergesort and bytecode. The solutions to these exercises are found in the section *Chapter 1*.

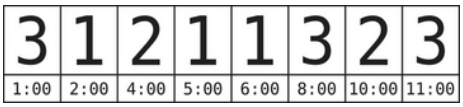
1. Say that we want to Mergesort the array `[1, 5, 2, 7, 4, 3, 6, 8]`. Fill in the chart to demonstrate how Mergesort recursively breaks down the array:



2. Let's continue with the example from the previous exercise. Fill in the next chart to show how Mergesort merges all the values to produce a completely sorted array:



3. *New Concept:* Each data element that follows consists of two pieces: the first (top) piece is the primary data itself—an integer—and the second (bottom) piece is a timestamp of when that integer was created. Currently, the array is sorted in terms of the timestamp, but now we want to use Mergesort to sort the array by the integers themselves:



   Use pencil and paper (or whatever) to walk through all the steps of Mergesort. What do you notice about the sorted array once you're done?

4. Write a Python loop that prints out the integers `1` through `10`. Then, produce the bytecode for your Python code. What do you get?

   Now, write a second version of the same program using a different type of loop (such as a `while` loop instead of a `for` loop, for example). Produce the bytecode for this version as well and compare the two sets of bytecode. What aspects of the two sets of bytecode seem similar, and what aspects seem different?

   There's no right or wrong answer to this exercise. It's just to give you practice with generating bytecode and (lightly) analyzing it.

Chapter 2

Benchmarking Code

Throughout Volume 1, we'd worked with the idea that an algorithm's speed is measured by counting its steps. However, you saw in the previous chapter that sometimes simply counting steps isn't enough to determine which algorithms are faster than others. For example, both Quicksort and Mergesort take roughly the same number of steps, yet computer scientists report that Quicksort is significantly faster in actual time. Again, by *actual time* I mean time measured in minutes and seconds rather than in steps.

How do the computer scientists know this? More importantly, how can *you* know which algorithms are truly the fastest?

Here enters a technique known as *benchmarking*. In this chapter, you'll learn what benchmarking is and how to benchmark Python code. Along the way, you'll also learn about nefarious benchmarking traps and how to avoid them.

Benchmarking

In theory, you can compare the actual-time speeds of two algorithms by running the code on your computer and using a stopwatch to see how long each algorithm takes. This is the essence of what benchmarking is, except that benchmarking is a lot more practical. Besides being tedious, using a stopwatch to measure the speed of code is error-prone. Also, what if the two algorithms both run in less than one second? You'd better have fast thumbs.

Thankfully, you can use benchmarking software to measure the time for you. The software tracks the precise time your code starts running and when it finishes. The benchmarking tool then tells you exactly how much time your code took to run.

So, there's nothing fancy about benchmarking; it's basically the computer running a stopwatch for you. However, it does come with a big bonus: it measures time very precisely, down to the microsecond. (A microsecond is *one-millionth* of a second.)

Before we move on, I can't emphasize enough that for the duration of this chapter, we will not be measuring speed in terms of the number of steps at all. When it comes to benchmarking, we'll always be measuring the *actual time* of an algorithm.

Using the timeit Module

One of the easiest ways to benchmark Python code is with Python's `timeit` module. In the words of Python's documentation: "This module provides a simple way to time small bits of Python code."

Usually, the `timeit` module is used from the command line to measure small snippets of code, but since we're going to use it to measure more complex algorithms, we're going to call the module from within an actual code file.

Let's start with a basic example to see how `timeit` works. In Python, you can create an array containing one million integers (`0` through `999999`) with the following code:

```
array = []
for i in range(1_000_000):
    array.append(i)
```

(Note that Python allows us to use underscores to make long numbers more readable.)

Let's use `timeit` to see how quickly this code runs. Here's the code that does this:

```
import timeit

test_code = '''
array = []
for i in range(1_000_000):
    array.append(i)
'''

print(timeit.timeit(stmt=test_code, number=1))
```

I'll explain what the code means shortly, but first, let's run it.

I've saved this code inside a file called `bench_first_example.py`. If I now run `python bench_first_example.py` from the command line, I get the following output:

```
0.121737003326
```

This is the number of seconds that it took my computer to run my array-generating code. Wow, it ran in a fraction of a second! Specifically, it ran in `0.121737003326` seconds, which is roughly one-eighth of a second.

When I change my code to generate an array that contains only 100 integers, `timeit` spits out this result:

```
1.50203704834e-05
```

If you don't look too carefully, this may seem to be saying that my code took about 1.5 seconds to run. But this would make no sense. Why would creating an array of 100 elements be much slower than creating an array containing 1,000,000 elements?

However, notice the `e-05` at the end of this output. This is scientific notation, and is a shorthand way of expressing this number:

```
0.0000150203704834
```

This is about 10,000 times faster than the `0.121737003326` seconds it took to generate 1,000,000 integers.

As a quick tip, if you want to convert scientific notation into "regular" notation, there's an easy way to do so. Here's an example:

```
print('%.08f' % 1.50203704834e-05)
```

The `.08` represents how many digits you want to see, which in this case is 8 digits. The previous code outputs this result:

```
0.00001502
```

If you want to see, say, 10 digits, you'd run this:

```
print('%.10f' % 1.50203704834e-05)
```

This outputs a result:

```
0.0000150204
```

Breaking Down the Code

Let's break down how the `timeit`-based code works.

First, we import the `timeit` module:

```
import timeit
```

The next part may seem a little wonky, but `timeit` demands that we pass in all of the code we're benchmarking *as a string*. Here, we store the string in a variable called `test_code`:

```
test_code = '''
array = []
for i in range(1_000_000):
    array.append(i)
'''
```

For readability, I used the triple-quote Python syntax for creating a multiline string. I also named the variable `test_code`, but it could be named anything you want.

We then pass our `test_code` into the `timeit` function and print the results:

```
print(timeit.timeit(stmt=test_code, number=1))
```

The `timeit` function can accept numerous options as parameters. The main argument is `stmt`, and that's where we pass in our `test_code`.

The `number` argument represents how many times we want to run the `test_code`. For this example, we'll run our `test_code` one time. Soon, I'll explain why you may want to run your code multiple times, but for now, we're going to keep `number` as `1`. However, I will point out now that if you don't pass in any `number` argument, the default is `1,000,000`! So, I always recommend passing in the `number` function. Otherwise, you may be waiting a *long* time.

Using timeit to Compare Two Algorithms

Measuring the speed of a single algorithm alone has limited value. Knowing that one algorithm takes, say, two seconds is somewhat meaningless since the same algorithm may run significantly faster or slower on other computers. So, no algorithm can ever be labeled a "two-second" algorithm. This is one of the reasons we've always counted steps as a more reliable way to express an algorithm's speed. After all, the number of steps that an algorithm takes remains consistent no matter which computer it's run on.

That being said, benchmarking shines when you are trying to compare two or more competing algorithms against each other. This is indeed a useful measurement since as long as you do all of the benchmarking on the same computer, you can find out which algorithm is the fastest in actual time.

Let's try this out for testing two of Python's built-in functions. In the previous example, we benchmarked using the `append` method to create an array of one million ascending integers.

Python has another method for adding values to an array, namely, the `insert` method. If we want to generate an array of one million integers with this method, we'd run the following code:

```
array = []
for i in range(1_000_000):
    array.insert(len(array), i)
```

With benchmarking, we can determine whether `append` or `insert` is a faster method for populating an array.

Here's my benchmarking code. It's virtually the same as our code for benchmarking `append`, except that I'm using the `insert` approach:

```
import timeit

test_code = '''
array = []
for i in range(1_000_000):
    array.insert(len(array), i)
'''

print(timeit.timeit(stmt=test_code, number=1))
```

When I benchmark the `insert` code, I get a result of:

```
0.353770017624
```

This is almost three times slower than our `append` benchmarking result, which was `0.121737003326`. Based on this benchmarking experiment, it would seem that `append` is notably faster than `insert`.

It can be super fun to conduct benchmarking experiments. We assume the role of scientists and measure real, tangible results. It's so … scientific!

Benchmarking Gotchas

While benchmarking at its core is a simple idea, a surprising number of gotchas can completely derail your experiments. These gotchas are particularly sneaky since `timeit` will always spit out a number even if your experiment isn't set up correctly. Everything may *seem* to be in order, but your results may be totally meaningless. We'll look at these gotchas throughout the remainder of this chapter.

Gotcha: Using Two Different Sets of Numbers

One of the most important things to keep in mind when benchmarking, or when conducting any scientific experiment for that matter, is to ensure that your experiment is *controlled*. This means that if you're comparing two different algorithms, all other factors besides the algorithms themselves should be identical.

Take our `append` vs. `insert` experiment, for example. Our experiment is only valid if both snippets of code are creating arrays of the same size. If, however, our `append` code created 1,000,000 elements while our `insert` code created only *100* elements, our benchmarking results would be skewed. The fact that our `append` code ran more slowly wouldn't prove that `append` is slower than `insert`; perhaps it ran more slowly only because it was busy generating so many more elements.

Indeed, when I run the `insert` code with just 100 elements, I get the supersonic result of `3.69548797607e-05`. This is way faster than the results of my `append` benchmark of one million elements, and if I wasn't paying attention, I might mistakenly conclude that `insert` is much faster than `append`.

Because of this, when benchmarking two competing code snippets, you always want to double-check that both algorithms are working with the same data.

Gotcha: Only Benchmarking One Time

Obviously, a benchmarking experiment would not be controlled if you tested Algorithm A on one computer and Algorithm B on another. Perhaps the only reason why one algorithm runs faster than the other is that it was executed on the more powerful computer!

Similarly, it's possible that even when you benchmark two algorithms on the *same* computer, the computer happens to be more powerful when executing one algorithm than when executing the other. This is very common since a computer is always running multiple processes at any given time. The benchmarking code you run is never executed in a vacuum. You may have other applications running, including that Internet browser with 85 tabs open. (You should probably do something about that.)

Based on this, it would be unwise to benchmark Algorithm A and then decide to play a massive multiplayer video game while benchmarking Algorithm B. Perhaps the results of Algorithm B are slower because the game is running in the background.

However, even if you don't turn on that game, you can't know for certain that your computer isn't secretly downloading some security update while you happen to be benchmarking Algorithm B. Basically, your computer is always running all sorts of processes in the background, and there's not much you can do about it.

One way to help with this is to run your benchmarking experiments multiple times. It's less likely that the same background process will keep occurring each time you run your experiment.

timeit's Repeat Method

The `timeit` module provides a method called `repeat` that will conveniently run your benchmark multiple times so you don't have to do it manually. The `repeat` method is essentially identical to the `timeit` method, except that `repeat`

also accepts a `repeat` argument where you set the number of times that your code should execute:

```
print(timeit.repeat(stmt=test_code, repeat=5, number=1))
```

I'll explain the difference between the `repeat` argument and the `number` argument soon, but for now, we'll focus on `repeat`.

With this code, our benchmark will run 5 times. This will return an *array* of results like these:

```
[0.3296499252319336, 0.2981288433074951, 0.3133430480957031,
0.3073868751525879, 0.3087730407714844]
```

These are the results of running my `insert` benchmark on 1,000,000 elements. The results are all similar, but the fastest among these results is `0.2981288433074951`.

Now, here's an important point. In theory, a piece of code should take the same amount of time each time we execute it. So why aren't the results all exactly the same? Again, this is because the computer's background processes will always skew the results somewhat.

Because of this, the truest of the results is, in fact, `0.2981288433074951`. The only reason why the other results were slightly slower than this is that the computer's background processes got in the way. So instead of using the *average* of our results to get the most accurate speed, professional benchmarkers use the *fastest* result.

However, this assumes that our code has no randomness involved. If our data is randomized each time we run it, the varying speeds may be a result of the fact that the data is different each time we run our benchmark. This will come into play when we benchmark sorting algorithms, as you'll soon see.

Repeat vs. Number

Both the `repeat` argument and the `number` argument allow you to execute your code numerous times in a row. However, there are two key differences between the two arguments.

One key difference is with regard to how the results are displayed. The `repeat` argument spits out an array of different results, as you saw earlier in this chapter. However, if we keep `repeat` at 1 and instead change `number` to 5, we get one result that is the amount of time that it took for *all* five rounds to execute *in total*. So, if I get the result of `1.56467604637146`, this means I would have to divide that number by 5 to see how fast each individual round took on average.

Whether you use `repeat` or `number`, you'll always want to benchmark Algorithm A and Algorithm B the same number of times to keep your experiment controlled.

In any case, I'll be using the `repeat` approach going forward, generally setting the `repeat` argument to `5`.

There's another key difference between `repeat` and `number`, but before we look at that, let me address another gotcha.

Gotcha: Not Making Sure Your Code Works

I once read an online tutorial on benchmarking that used this example code:

```
import timeit

test_code = '''
def create_massive_array():
    array = []
    for i in range(1_000_000):
        array.append(i)
'''
```

```
print(timeit.repeat(stmt=test_code, repeat=5, number=1))
```

It's basically the same `append` experiment we used earlier, except that all the code is wrapped inside a function called `creative_massive_array`.

When I benchmark this code, I get very strange results:

```
[1.1920928955078125e-06, 0.0, 0.0, 9.5367431640625e-07, 0.0]
```

What on Earth? What's with all the zeroes? And why are all of these numbers so different from each other?

Before I reveal the solution, I want to highlight that this should be your most important takeaway from this chapter: *always keep your brain on*. If you see surprising results, you shouldn't blindly accept them. Instead, investigate *why* you're getting those results. If something smells fishy, it probably is.

One of the first steps to take when discovering fishy results is to make sure your code does what you think it should do. For this, printing to the console is your friend. Let's go ahead and add a `print(array)` command at the end of our `test_code`:

```
test_code = '''
def creative_massive_array():
    array = []
    for i in range(1_000_000):
        array.append(i)
    print(array)
'''
```

When I benchmark this revised code, no array gets outputted to the console. I still get wonky benchmark numbers, but I should have also seen a massive array displayed in the console. What's going on? Wait … facepalm emoji. (Is that a phrase?) The code for generating the array never gets executed! Sure, my `test_code` *defines* a function that will generate an array, but this function never gets called anywhere. No wonder this code ran so fast.

As to why we got those strange benchmarking numbers, that has to do with Python internals. If you benchmark any code that hardly does anything, you'll get similar wonky results.

To avoid this gotcha, it's worthwhile to always use `print` or a similar technique to ensure that your code is doing what it should.

To fix this particular problem, we need to *call* the function within our `test_code` itself:

```
test_code = '''
def create_massive_array():
    array = []
    for i in range(1_000_000):
        array.append(i)
    print(array)
create_massive_array()
'''
```

After we've verified that our code works, we can then eliminate the `print` statement and benchmark our code properly. In fact, we should indeed make a point of removing the `print` statement. This is because printing to the console consumes considerable time in its own right, and we only want to benchmark the actual algorithm, not the printing.

Benchmarking Sorting Algorithms

This is the fun part—benchmarking our sorting algorithms! I saved our Mergesort code from *Code Implementation: Mergesort* inside a file called `mergesort_1.py`. The strategy we'll follow to benchmark our code is to first generate an array of integers in *random* order and then benchmark how quickly Mergesort can sort the array. (The following code contains some new elements we haven't encountered yet, but I'll walk through them shortly.)

```
import timeit

setup_code = '''
import random
import mergesort_1

array = []
for i in range(10):
    n = random.randint(1, 1000)
    array.append(n)
'''

test_code = '''
mergesort_1.mergesort(array)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5, number=1))
```

As usual, we begin by importing the `timeit` module. However, the code is divided into two sections. In addition to our `test_code`, we now also have `setup_code`. Let's take a look at what the `setup_code` is all about.

To test out Mergesort, we first generate an array containing integers in random order and then perform Mergesort on that array. Technically, we could have put all of our code inside our `test_code` string. However, our true goal is to benchmark the Mergesort algorithm alone. Generating the random

array is *setting things up* for Mergesort to do its work. As such, we don't care to measure how long it takes to generate the array; it's just *setup* code.

This is why `timeit` allows us to put our "setup code" inside a separate string, which we brilliantly named `setup_code`. This is excluded from the benchmark itself. That is, `timeit` will only measure the running time of the `test_code`, and not the `setup_code`.

As you can see, we pass the `setup_code` into an argument `setup` inside the `repeat` method:

```
print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5, number=1))
```

However, there's a sneaky gotcha with setup code that'll bite you if you're not careful.

Gotcha: Accidentally Running the Setup Code Just Once

I mentioned earlier that there are two major differences between the `repeat` and `number`. We already looked at the first difference, and now we'll look at the second, which is hugely important.

If you pass in the arguments `repeat=5, number=1`, the `timeit` method will run both the setup code and the test code 5 times. However, if you did the opposite and passed in `repeat=1, number=5`, the setup code is executed *only once*, followed by the `test_code` running 5 times in a row.

This may not matter in some cases, but it can certainly matter when benchmarking sorting algorithms. That is, if the setup code is executed only once, *the array is completely sorted after the first run*. When the computer executes the sorting algorithm the next 4 times, we're sorting an array that's already completely sorted!

Mergesort takes the same amount of time whether the array is already sorted or not, but other sorting algorithms like Insertion Sort run way faster

when the array is already sorted. On the flip side, as you saw in the previous chapter, Quicksort runs much *slower* when the array is already sorted. So, if we only run our setup code once, our benchmarking results are going to be skewed. We need to make sure that on each run, we regenerate a randomly sorted array.

Mergesort vs. Insertion Sort

When I benchmark Mergesort using the code in *Benchmarking Sorting Algorithms* for an array of 10 elements, I get these results:

```
[2.6941299438476562e-05, 2.2172927856445312e-05, 2.193450927734375e-05,
2.193450927734375e-05, 2.193450927734375e-05]
```

As I've said, these numbers aren't very meaningful until we contrast them with the benchmarking results of a competing algorithm. So, let's go ahead and benchmark Insertion Sort and compare the two sets of results.

Before we do so, it's always good to hypothesize as to what the results might be. This way, if we get totally different results, we can more easily notice if something fishy is going on and decide if our experiment isn't engineered correctly.

You learned in the previous chapter that Mergesort runs in O(N log N) time. And as discussed in Volume 1, Chapter 6, Insertion Sort on average takes $O(N^2)$ time. It's reasonable to hypothesize that Mergesort should be the faster of the two algorithms. With this guesstimate in mind, let's benchmark Insertion Sort.

Here's the Insertion Sort code from Volume 1, which I saved in a file called **insertion_sort.py**:

```
def sort(array):
    for i in range(1, len(array)):
        key_item = array[i]
        j = i - 1

        while j >= 0 and array[j] > key_item:
            array[j + 1] = array[j]
            j -= 1

        array[j + 1] = key_item
```

```
    return array
```

And here's the benchmarking code for Insertion Sort:

```
import timeit
setup_code = '''
import random
import insertion_sort

array = []
for i in range(10):
    n = random.randint(1, 1000)
    array.append(n)
'''

test_code = '''
insertion_sort.sort(array)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5, number=1))
```

When I run this code, I get the following results:

```
[1.811981201171875e-05, 1.4781951904296875e-05, 1.4066696166992188e-05,
1.5974044799804688e-05, 1.5020370483398438e-05]
```

Whoa! These results are *faster* than our Mergesort results. That's not what I expected at all.

Now, I did run the Mergesort benchmark several minutes ago, and who knows what background processes were running then compared to now. So, I'm going to run both benchmarks again to keep this experiment as controlled as possible. We've benchmarked Insertion Sort, so let's benchmark Mergesort again.

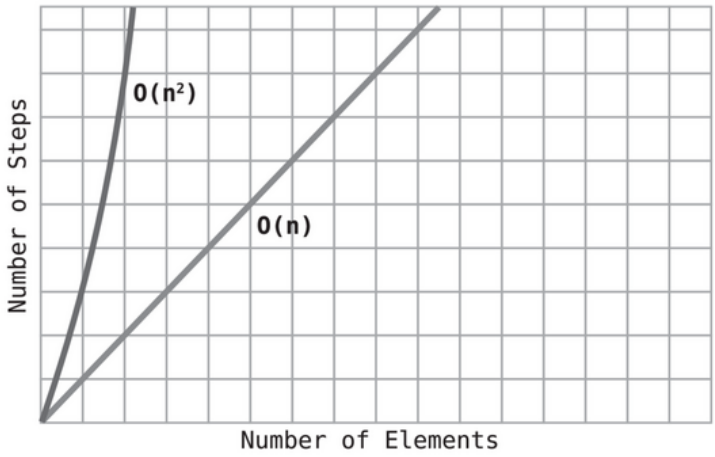
My Mergesort results now are these:

```
[2.3971296527821164e-05, 2.2451671856349302e-05, 2.190422274014396e-05,
2.195827825732071e-05, 2.193578247516832e-05]
```

Hmmm, I'm getting similar results as before. So, what's wrong with my experiment? Why does it seem that Insertion Sort is faster than Mergesort?

Gotcha: Only Using One Data Size

As discussed in Volume 1, Chapter 3, we use Big O notation to measure the *trajectory* of an algorithm as the data grows. The difference between each category of Big O is, in fact, much more noticeable as the size of the data grows. Take the following diagram:



Here, for example, we compare the trajectories of $O(N^2)$ vs. O(N). It's true that as the data increases, O(N) becomes faster compared with $O(N^2)$. However, when we have a small amount of data, as shown on the left side of the graph, the speeds of the two algorithms aren't all that far apart.

Because of this, when benchmarking two competing algorithms, you also want to run the benchmarks on *larger amounts of data*. The larger the amount of data, the greater the discrepancy between the two sets of results will be, generally speaking.

When I change the benchmarking code to have both Mergesort and Insertion Sort operate on an array of size `10_000` (instead of `10` as before), I get these results:

```
Mergesort:
```

```
[0.05725693702697754, 0.055114030838012695, 0.06000399589538574,
0.057826995849609375, 0.056210994720458984]

Insertion Sort:
[4.285884141921997, 4.154591083526611, 4.248677015304565,
4.378739833831787, 4.4347240924835205]
```

That's a huge difference, and much more in line with our hypothesis. However, a question may still be tugging at your brain.

Benchmarks for the Large and Small

When the array was of size 10, it's understandable that both Mergesort and Insertion Sort had similar speeds, but why was Insertion Sort *faster* than Mergesort? Insertion Sort runs at $O(N^2)$, which means that it should take about 100 steps. Mergesort, on the other hand, is O(N log N). N is 10, and log N is about 3, so we're talking about roughly 30 steps.

This gets back to the reason we began benchmarking in the first place. Counting the number of steps is certainly important, but it's not the only factor that determines an algorithm's actual-time speed. On the machine-code level, Mergesort's 30 Python steps have more "overhead" than Insertion Sort's 100 Python steps.

While I won't get into the details of how our two sorting algorithms operate on a machine-code level, the following analogy should suffice:

If we asked both an experienced weightlifter and a couch potato to perform 100 pushups, the weightlifter would likely complete them much more quickly than the couch potato. However, knowing the importance of warming up before a workout, the weightlifter might take a few minutes to stretch and do some jumping jacks. Now, even if the couch potato doesn't do a warmup, the weightlifter will *still* likely complete all 100 pushups before the couch potato.

However, if we asked each person to do 10 pushups, the couch potato might complete them first since the weightlifter may still insist on warming up before doing *any* physical activity. By the time the weightlifter finished warming up, the couch potato might already be done (if he's lucky).

Here as well, Mergesort uses tools such as recursion and copying arrays to perform its powerful work. However, these tools carry a certain amount of overhead that takes some time. Of course, it's certainly worth using these tools so that Mergesort can sort an array of size 10,000 in O(N log N) time. But when it comes to small arrays, Insertion Sort, which doesn't have all this overhead, can complete the job *quicker* than Mergesort despite Insertion Sort being the "slower" algorithm.

Optimizing Mergesort with Insertion Sort

You've seen that it's a mistake not to use large data when benchmarking. However, it's also a mistake not to use *small* data when benchmarking.

We learn this lesson from Insertion Sort. Had we never benchmarked Insertion Sort for an array of size 10, we may never have discovered that Insertion Sort is faster than Mergesort for small arrays.

At first glance, this novelty may seem unimportant. After all, we don't generally care how fast a sorting algorithm works on a small array since even the notoriously slow Bubble Sort algorithm (discussed in Volume 1, Chapter 4) sorts a small array quickly in actual time. However, computer scientists have discovered that we can utilize this attribute of Insertion Sort for the sake of *optimizing Mergesort*.

To do this, we employ a brilliant little trick: we modify the Mergesort algorithm so that when it encounters an array that has 10 elements or fewer, *it switches to Insertion Sort for that array*. We've never seen one sorting algorithm switch to another midstream, but I assure you that it's not illegal.

To implement this in our code, we only have to make the tiniest change to our `mergesort` function. I placed the `insertion_sort` function inside the same file as my `merge` and `mergesort` functions and then swapped out the base case:

```
def mergesort(array):
    if len(array) <= 10:
        insertion_sort(array)
        return

    midpoint = len(array) // 2
    copy_of_left_half = array[:midpoint]
    copy_of_right_half = array[midpoint:]

    mergesort(copy_of_left_half)
    mergesort(copy_of_right_half)
    merge(copy_of_left_half, copy_of_right_half, array)
```

Instead of the base case being an array of size 1, the base case is now an array of size less than or equal to 10. When this new base case is encountered, we call `insertion_sort` on that base-case array.

Let's now benchmark regular Mergesort against this optimized Insertion Sort-infused version of Mergesort. For an array of size `1_000_000`, I get these results:

```
Regular Mergesort:
[8.321918964385986, 8.336308002471924, 9.427529096603394,
9.48000192642212, 9.64377498626709]

Optimized Mergesort:
[6.830552101135254, 6.836566925048828, 7.844353914260864,
7.90967321395874, 7.877916097640991]
```

The optimized version shaves off about one to two seconds from Mergesort's runtime. With larger amounts of data, this optimization would be even more significant.

In our implementation, we used the number 10 as the tipping point for when Mergesort switches over to Insertion Sort. However, 10 isn't necessarily *the* magic number. To determine what the number should be, you could keep benchmarking different numbers until you find the one where Mergesort and Insertion Sort are roughly the same speed. In truth, though, the number may depend on the particular computer you're using, so there may not be a definitive number.

Another interesting thing to note is that when it comes to small arrays, Insertion Sort isn't only faster than Mergesort; it's faster than Quicksort, too. That's why many finely tuned implementations of Quicksort also switch over to Insertion Sort when it encounters a small array.

Mergesort vs. Quicksort

This entire discussion of benchmarking came about as a result of contrasting Mergesort with Quicksort in the previous chapter, so let's use benchmarking to see for ourselves whether Quicksort is truly the faster algorithm.

Following is the Quicksort code from Volume 1, Chapter 13:

```
class SortableArray:

    def __init__(self, array):
        self.array = array

    def partition(self, left_pointer, right_pointer):
        pivot_index = right_pointer
        pivot = self.array[pivot_index]
        right_pointer -= 1

        while True:

            while self.array[left_pointer] < pivot:
                left_pointer += 1

            while self.array[right_pointer] > pivot:
                right_pointer -= 1

            if left_pointer >= right_pointer:
                break
            else:
                self.array[left_pointer], self.array[right_pointer] = \
                    self.array[right_pointer], self.array[left_pointer]
                left_pointer += 1

        self.array[left_pointer], self.array[pivot_index] = \
            self.array[pivot_index], self.array[left_pointer]

        return left_pointer
```

```
    def quicksort(self, left_index, right_index):
        if right_index - left_index <= 0:
            return

        pivot_index = self.partition(left_index, right_index)

        self.quicksort(left_index, pivot_index - 1)

        self.quicksort(pivot_index + 1, right_index)
```

Here is my Quicksort *benchmarking* code. For this experiment, I'll time how long it takes to sort one million random integers:

```
import timeit

setup_code = '''
import random
import quicksort

array = []
for i in range(1_000_000):
    n = random.randint(1, 1_000_000)
    array.append(n)

sortable_array = quicksort.SortableArray(array)
'''

test_code = '''
sortable_array.quicksort(0, len(array) - 1)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5, number=1))
```

Since it's been a few minutes since benchmarking Mergesort, it's best to run that again, too. Note that I'm benchmarking our "regular" Mergesort, not the one we optimized using Insertion Sort. Here are my side-by-side results:

```
Mergesort:
[9.667517900466919, 9.65391206741333, 11.16753602027893,
10.575589895248413, 10.170050144195557]

Quicksort:
```

```
[5.473771095275879, 5.405587196350098, 5.787222862243652,
5.61734414100647, 6.785470008850098]
```

Wow, Quicksort is almost twice as fast. That's pretty impressive, and pretty much in line with what computer scientists have been telling us. Go computer scientists!

The moral of the story is that although two algorithms may take the same number of Python steps, one can be significantly faster than the other in actual time. Again, this is due to factors under the hood of the computer, such as the number of machine-code steps and spatial locality. These under-the-hood factors may be difficult to predict, and that's exactly why we use benchmarking to determine the actual-time speed of an algorithm.

Using Python's Built-In Sorting Algorithm

Out of curiosity, let's benchmark Python's built-in `sort()` method for arrays. Here's my benchmarking code for timing the sorting of one million random integers:

```
import timeit

setup_code = '''
import random

array = []
for i in range(1_000_000):
    n = random.randint(1, 1_000_000)
    array.append(n)
'''


test_code = '''
array.sort()
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5, number=1))
```

Here are my results:

```
[0.7919449806213379, 0.7905678749084473, 1.1442229747772217,
1.0725150108337402, 1.0439801216125488]
```

Pondering the meaning of this as I climb back into my chair, it appears that it almost always pays to use the sorting algorithm built into the language you're using. This is *way* faster than our Mergesort and Quicksort implementations, and even that's an understatement.

As of this writing, Python uses a sorting algorithm called Timsort, named after Tim Peters, who first implemented it. Timsort uses both merging and Insertion Sort together with some other techniques. I encourage you to check it out.

Now, the reason why a language's built-in sorting algorithm is so fast isn't only because it uses Timsort. Whenever any language's built-in methods are implemented, every optimization trick in the book is used to ensure that the method is as fast as possible. As such, even if a language uses Quicksort under the hood of its built-in sorting algorithm, it's likely to be faster than a textbook implementation.

Quicksorting a Sorted Array

In the previous chapter, it appeared that possibly the most significant advantage of Mergesort over Quicksort is that Mergesort runs at O(N log N) in all scenarios, while Quicksort is $O(N^2)$ in the worst case. Again, for Quicksort, the worst case is an array that's already sorted.

It would be nice to see how this plays out in actual time, so let's benchmark it. Here's our code for benchmarking Quicksort on a sorted array of size 10,000:

```
import timeit
setup_code = '''
import quicksort

array = []
for i in range(10000):
    array.insert(0, i)
sortable_array = quicksort.SortableArray(array)
'''

test_code = '''
sortable_array.quicksort(0, len(array) - 1)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5, number=1))
```

Welp, I don't get any results at all. Instead, I receive this error message:

```
RuntimeError: maximum recursion depth exceeded
```

This makes sense, come to think of it, since Quicksort calls itself recursively and the call stack doesn't unwind until we reach the base case of the left and right pointers meeting. And when the array is sorted, this won't happen until we're thousands of calls deep.

To get the code to complete on my computer, I have to change the array size to a paltry 900. When I benchmark both Mergesort and Quicksort on a sorted array of this size, I get the following results:

```
Mergesort:
[0.0037310123443603516, 0.0049169063568115234, 0.003873109817504883,
0.0034210681915283203, 0.0033721923828125]

Quicksort:
[0.04238104820251465, 0.04095888137817383, 0.06446003913879395,
0.087677001953125, 0.06192493438720703]
```

Indeed, Mergesort in this scenario is at least 10 times faster.

This confirms what we saw earlier: while Quicksort is faster than Mergesort in the average case, Mergesort is faster than Quicksort in a worst-case scenario. This would seem to be Mergesort's redeeming quality.

But there's bad news for Team Mergesort. There's an optimization for Quicksort that will ensure that it, too, will not slow down in a worst-case scenario, which thereby eliminates Mergesort's signature advantage. In the next chapter, we'll explore this optimization, which in turn will unlock an entire class of algorithms and data structures that will become the foundation for the rest of this book.

Wrapping Up

Benchmarking is a powerful technique that measures code execution speed in a fine-grained way, allowing you to know the true, actual-time difference between competing algorithms. This is especially useful when you want to compare two algorithms that fall within the same category of Big O. However, benchmarking can even reveal surprises about algorithms that *aren't* in the same category. As you discovered in this chapter, in some cases, the $O(N^2)$ Insertion Sort algorithm can be faster than O(N log N) Mergesort!

On the other hand, you also learned that benchmarking is only useful when used *correctly*. There are numerous gotchas that can derail your experiments without you even realizing it. Therefore, it's worthwhile to periodically look back at the gotchas in this chapter to make sure you're setting up your benchmarking code properly.

In the next chapter, we'll continue with our analysis of Mergesort vs. Quicksort. And in doing so, we'll stir things up once again, this time with the concept of *randomization algorithms*.

Exercises

The following exercises provide you with the opportunity to practice with benchmarking Python code. The solutions to these exercises are found in the section *Chapter 2*.

1. Following are two different Python functions that accept an unsorted array of integers and return the smallest number from the array. Here's the first version:

```
def minimum(array):
    smallest_item_so_far = float('inf')

    for item in array:
        if item < smallest_item_so_far:
            smallest_item_so_far = item

    return smallest_item_so_far
```

   This first version performs a linear search on the array while keeping track of the smallest number throughout the search.

   Here's the second version:

```
def minimum(array):
    array.sort()
    return array[0]
```

   This second version accomplishes the same task of returning the smallest number using another approach. It first sorts the array by ascending order and then returns whichever item is at the beginning of the array. Naturally, this item will be the smallest value.

   Hypothesize which version you think will run faster. Then, write and run benchmarking code to confirm whether your hypothesis is correct.

2. Following are two functions that both sum up all integers from `1` up until (but not including) `1_000_000`.

   The first version uses a `for..range` loop:

   ```
   def sum_up_to_one_million():
       sum = 0

       for i in range(1_000_000):
           sum += i

       return sum
   ```

   The second version uses a `while` loop:

   ```
   def sum_up_to_one_million():
       sum = 0
       i = 1

       while i < 1_000_000:
           sum += i
           i += 1

       return sum
   ```

   Hypothesize which function you think will run faster. Then, write and run benchmarking code to confirm your hypothesis.

3. Following is code that benchmarks an awesome new sorting algorithm that I've just invented. I call it `awesome_sort`, and it's too awesome for me to even show you how it works. However, I *will* show you my benchmarking code that I'm using to test it.

   But there's a problem, as my benchmarking code is not set up correctly. Can you spot the mistake?

   ```
   import timeit
   import awesome_sort
   ```

```
setup_code = '''
array = []
for i in range(100_000):
    array.append(i)
'''

test_code = '''
awesome_sort.sort(array)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5,
number=1))
```

4. I've always wanted to use benchmarking to discover the actual-time performance difference between linear search and binary search. However, this time I've made *two* mistakes in my benchmarking code. Can you find them?

   Here's my code for benchmarking linear search:

```
import timeit

setup_code = '''
def linear_search(array, search_value):
    for index, element in enumerate(array):

        if element == search_value:
            return index
        elif element > search_value:
            break

    return None

array = []
for i in range(100_000):
    array.append(i)
'''

test_code = '''
print(linear_search(array, 89124))
'''
```

```
print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5,
number=1))
```

And here's my code for benchmarking binary search:

```
import timeit

setup_code = '''
def binary_search(array, search_value):
    lower_bound = 0
    upper_bound = len(array) - 1

    while lower_bound <= upper_bound:

        midpoint = (upper_bound + lower_bound) // 2
        value_at_midpoint = array[midpoint]

        if search_value == value_at_midpoint:
            return midpoint
        elif search_value < value_at_midpoint:
            upper_bound = midpoint - 1
        elif search_value > value_at_midpoint:
            lower_bound = midpoint + 1

    return None

array = []
for i in range(1_000_000):
    array.append(i)
'''

test_code = '''
binary_search(array, 89124)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5,
number=1))
```

Chapter 3

How Random Is That?

In the previous two chapters, we encountered a notable trade-off between Mergesort and Quicksort. On the one hand, Quicksort is the faster algorithm in average-case scenarios. On the other hand, Mergesort guarantees speedy sorting in *all* scenarios, while Quicksort slows down for arrays that already happen to be sorted. The trade-off is this: would you rather use an algorithm that is super fast in most scenarios but sometimes slow or an algorithm that is only moderately fast but guarantees that it'll never be slow?

In this chapter, you'll discover that this is a false dichotomy. We can use a technique called *randomization* to perform Quicksort so that it's faster than Mergesort in *all* scenarios. You'll likewise learn how randomization can play an important role in optimizing data structures and algorithms by greatly improving an algorithm's speed. And perhaps most fun of all, you'll get insight into how a computer performs randomization in the first place, from generating random numbers to shuffling arrays.

Randomized Quicksort

As mentioned in *Trade-Offs*, the worst-case scenario for Quicksort is an array that's already sorted. But there's an easy way to fix this with a concept that is both ridiculously simple and yet counterintuitive at the same time: before executing Quicksort, you can first *shuffle the array*.

To shuffle an array means to put its values in random order, similar to shuffling a deck of cards. In Python, you can shuffle an array using the built-in `random` module:

```
import random

array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(array)
```

This modifies `array` by randomizing the order of its values.

I mentioned that this technique is both simple and counterintuitive at the same time. It's simple because it attacks the problem head-on. Quicksort is slow when processing a sorted array, so we simply *un*sort the array before running the rest of the algorithm.

Yet, it's also counterintuitive since who would have thought to speed up a sorting algorithm by first *shuffling* the values?

In truth, though, the real counterintuitive idea here is that Quicksort is slow for an array that's already sorted. One would have thought that we'd have to *less* work to sort such an array, not *more* work. However, this is the reality, so shuffling the array does the trick of ensuring that Quicksort will never be slow for any scenario.

I should point out that even shuffling an array isn't a 100 percent *guarantee* that Quicksort won't be slow. It is theoretically possible that shuffling an

array will produce the same array again! However, the odds that this might happen are low, and the odds sink further as we deal with larger and larger arrays. The more data there is, the less chance there is that shuffling an array ends up producing the same array again.

Now, shuffling the array does take some time. We'll analyze how many steps it takes, but take my word for it that preshuffling an array before Quicksort is a lot faster than Quicksorting a presorted array.

If you're always going to preshuffle the array before performing Quicksort, you can view that as a new step within the Quicksort algorithm. You're essentially creating a variant of the classic Quicksort algorithm, which doesn't necessarily preshuffle the array.

As a matter of fact, preshuffling isn't the only way to use randomization within Quicksort. Another approach is to always choose a random element to serve as the pivot rather than the left- or right-most element. I'm not going to explain this in depth since I don't want to get back into the weeds of Quicksort right now. However, if you're interested, the gist of this approach is as follows:

The reason why Quicksort performs badly on a sorted array is that when the array is sorted, the left- or right-most pivot will always end up at the array's end rather than the center. But by choosing a random pivot, the pivot has a similar chance of landing toward the center whether the array is sorted or not. Again, this explanation will make sense if you recall the details of Quicksort, which you can find in Volume 1, Chapter 13.

In any case, we now have variants of Quicksort to ensure that the sorting will take place in O(N log N) time for *all* scenarios. Computer scientists refer to these variants as *Randomized Quicksort*. (It's nice when computer scientists give an algorithm a name that makes sense.)

When we now compare Mergesort against Randomized Quicksort, it appears that Mergesort no longer has any advantage. The one potential advantage was that Mergesort doesn't slow down in worst-case scenarios, but now Randomized Quicksort doesn't either.

We're still not done pitting these two sorting algorithms against each other, but let's first spend some time exploring the concept of randomization. In fact, randomization is going to emerge as one of the main themes of the rest of this volume.

Randomized Algorithms

Randomization has broader ramifications than being cleverly used to speed up Quicksort. Randomized Quicksort is one example of an entire class of algorithms known as *randomization algorithms* (or *randomized algorithms*).

The term *randomization algorithm* is a fancy name for an algorithm that uses randomization somewhere as part of its instructions. Randomized Quicksort is such an algorithm since its first step is to use randomization to shuffle the array. Randomized algorithms serve as a major theme throughout this book since they have so many applications.

As you'll discover throughout our journey, randomization can be used to perform all sorts of optimizations. With Randomized Quicksort, you saw how randomization improved the speed for worst-case scenarios. Some other randomization algorithms, though, use randomization to increase speed in *all* scenarios. There are algorithms that use randomization to save memory, while others use randomization to increase accuracy. (You'll see what that means in Chapter 9, *Counting on Monte Carlo Algorithms*.) And sometimes, randomization is at the core of the algorithm itself, and without randomization, the algorithm wouldn't work at all.

Before we explore other types of randomization algorithms, we should first answer some fundamental questions about randomization in general. For example, how does a computer shuffle an array? And what is the time complexity of shuffling an array?

For that matter, let's start with an even more basic question: how does a computer choose a random number?

Generating Random Numbers

If you've used Python, or any high-level programming language for that matter, it's easy to take for granted that you can use it to generate random numbers. For example, you can print a random integer from 1 through 10 with the following snippet:

```
import random

print(random.randint(1, 10))
```

But here's a fundamental, almost philosophical, question: how can a computer choose a random number? Here's what I mean.

Let's talk about randomness as it appears in the world. If I ask my friend to choose a random number between 1 and 10, is the number they choose truly random? Perhaps there's some psychology involved in the number they choose. In fact, some studies show that people choose 7 more often than any other number. While there's debate as to why that's so, it indicates that a human choosing a random number is not as random as we might think. After all, if the process was truly random, 7 should be as common a choice as any other number.

Let's consider possibly the most "classic" of all random activities: flipping a coin. Is it truly random to flip a coin to see whether it lands on heads or tails? In theory, physicists should be able to predict the result by measuring the strength one used to flip the coin, plus the weight of the coin, plus the air pressure, plus the hardness of the surface on which the coin lands, plus a few other factors. Indeed, if we want to get *really* philosophical, we can wonder whether *anything* in the world is truly random.

Despite these arguments, we can still define randomness in a way that is good enough for our purposes.

Here's the deal. Sure, it might be technically possible to predict the outcome of a coin toss. However, it's *really* difficult to measure all the relevant factors—and you and I certainly aren't going to do so. Therefore, we can consider flipping a coin to be *unpredictable enough* to be called random. And so, we can say that an event is random if *it's difficult to predict the outcome*.

But now let's get back to computers. A computer doesn't flip coins and certainly doesn't use original thinking to generate a random number "off the top of its head." A computer is what's called *deterministic*, meaning that it's only capable of receiving and executing instructions we give it. So, if we give the computer an algorithm to generate a random number, say, using mathematical calculations, we'll still be able to predict exactly what number the computer will choose if we know the algorithm. It won't be random at all.

This is a real head-scratcher. Is it even *possible* for a computer to choose a random number?

It turns out that there are two different ways in which a computer generates random numbers: with a True Random Number Generator (TRNG) or a Pseudorandom Number Generator (PRNG). We'll take a look at both, starting with TRNG.

True Random Number Generators

A TRNG uses a highly unpredictable external source rather than an algorithm to provide a random number. That is, since a computer is completely deterministic and cannot come up with a random number on its own, the computer relies on this outside source to do so.

This outside source could be anything in nature or the environment that is so hard to predict that we consider it random. For example, we could connect the computer to a secret office in a basement in South Dakota (the

epicenter of secret basement offices). Each time the computer needs a random number, it sends a signal to that office, and someone there will roll a die, which we'd consider a random event. The number that the die lands on is then sent back to the computer.

Of course, this is completely impractical. However, here are some TRNGs that are used in real life. In truth, some of them may sound almost as ridiculous as our South Dakota example, yet they are truly used in the real world.

- You know how if you set a radio to the wrong channel, you hear nothing but static noise? This static is generated by lightning strikes and other natural processes. One TRNG technique is to connect the computer to a radio that's set to a channel that picks up this static. The computer then picks out the frequency of sound at a precise moment in time and uses that frequency to generate the random number. As crazy as this sounds, this is currently one of the more popular TRNGs out there.

- Another real-life TRNG approach is to hook a computer up to a camera that takes a snapshot of a *lava lamp*, and generates a random number based on the exact formation of the "lava" at that given moment. I kid you not.

- Yet another TRNG connects the computer to *decaying radioactive material*. It's hard to predict the exact way in which an atomic nucleus decays, so why not use that for generating random numbers? (Don't try this at home.)

These are considered *true* random number generators because the random numbers are based on processes that are too difficult to predict. They're the best source for generating the "randomest" numbers.

However, there are two major disadvantages to TRNGs. First, they're not always practical, as you can imagine. Second, it can be relatively slow to generate a random number based on an external process, at least compared with PRNGs, which we'll look at soon.

Interestingly, there exist online TRNG services that allow you to connect your code to their TRNG devices via a web API. That is, *they* have computers hooked up to lava lamps or radioactive material or whatever, and all you have to do is connect to their servers and grab a truly random number that *they* generate. While this may help with the practicality issue, making a web request is way slower than having your computer generate a random number internally.

Because of these issues of practicality and speed, Python uses a PRNG rather than a TRNG. On that note, let's look at how PRNGs work.

Pseudorandom Number Generators

My dictionary tells me that the prefix *pseudo* means "false, not genuine, a sham." Indeed, pseudorandom numbers are just that—a total sham. They're not random at all; they only *seem* random. Yet, this is exactly the approach that Python and most other programming languages use to generate "random" numbers. Let's see how this works.

Prepared Sequence

The main idea behind PRNGs is that they have a *prepared sequence* of numbers that, at first glance, seem random. For example:

```
[5, 9, 1, 0, 2, 3, 4, 8, 0, 7, 2, 0, 0, 4, 9, 7, 0, 5, 0, 2]
```

When looking at these numbers at a glance, they do look kind of random. At least, there's no predictable pattern.

With a PRNG, each time we ask the computer to generate a "random" number, it will use this sequence. The first time we execute code that

generates a random number, the computer will return the first number from the sequence. In our example, this would be the number `5`. The *next* time we ask the computer to generate a random number, it will return `9` since it's the next integer in the sequence. And the third time we ask for a random number, the computer will return `1`. You get the idea.

When the computer reaches the end of the sequence (the final `2` in our example), it will then start all over again at the beginning of the sequence. There are 20 integers in the example sequence, so when we generate a random number for the 21st time, we'll get back a `5` since it's the first number in the sequence.

If you're horrified at this algorithm for generating random numbers, you should be. This isn't random at all! This approach is entirely deterministic; we know *exactly* what the computer will "randomly" choose next. Even worse, there are major problems with this sequence of numbers. For one thing, there are significantly more instances of `0` than any other number. And did you notice that there are no `6`s at all?

Now, we can improve things a bit if we create a better sequence. Here are a few ideas that are crucial for a decent sequence:

- One integer shouldn't be considerably more prevalent than the other integers. Each integer should appear the same number of times or at least close to it. This attribute is known as *uniformity*.

- The numbers should be "mixed up." If the sequence were `1`, `2`, `3`, `4`, and so on, it wouldn't even *seem* random. Even though pseudorandom numbers are, by definition, not truly random, we could at least try to make them look random. We call this attribute *lacking pattern*. That is, the sequence should lack any discernible pattern.

- The sequence should have a lot more integers. The previous example has only 20 integers. Even if there's no pattern within the 20 numbers

themselves, there still ends up being a pattern in that the same 20 sequence numbers keep repeating themselves! After a while, a user of our software may begin to notice this pattern. But what if we had 100 integers? That would be a more difficult pattern to detect. The number of integers in the sequence before it begins again is called the *period*. A sequence with a *long period* is one that has many numbers in it, so it'll take a while until the sequence starts over from the beginning again.

With this in mind, here's a better-looking sequence:

```
[4, 7, 2, 0, 0, 1, 8, 5, 1, 8, 3, 0, 7, 4, 7, 0, 3, 9, 2, 6,
 9, 1, 3, 3, 8, 9, 5, 4, 3, 1, 6, 8, 2, 9, 2, 0, 6, 5, 5, 9,
 6, 8, 5, 2, 2, 9, 7, 9, 8, 7, 6, 6, 0, 8, 5, 5, 7, 7, 4, 4,
 9, 8, 1, 7, 0, 3, 6, 1, 1, 4, 1, 9, 0, 9, 4, 2, 1, 5, 6, 4,
 2, 7, 3, 3, 6, 0, 6, 7, 2, 5, 4, 5, 3, 1, 8, 4, 8, 2, 3, 0]
```

This pattern contains 100 numbers, lacks pattern, and is uniform. Is it perfect? Far from it. But it's a lot better than our first sequence.

As contrived as this all seems, this is *pretty much how PRNGs work*. Since a computer can't generate a true random number, it fakes it. But it fakes it well enough to make the numbers seem random to the user.

Linear Congruential Generators

The concept of pseudorandom number generation is starting to take shape. But here's a new monkey wrench: so far, we've only been dealing with random numbers from 0 through 9. But what if we want to generate a number that was, say, 7 digits long?

Indeed, Python's `random()` method typically produces a number with a whole bunch of digits. Interestingly, the number generated is a float between 0 and 1. However, it has many digits beyond the decimal point. For example:

```
>>> random.random()
0.5339767180649452
```

And herein lies the problem. If we want uniformity to the point where every possible float occurs at least once within our sequence, we'd need to prepopulate a sequence containing *trillions* of numbers. That's a lot of memory to take up to enable a programming language to generate a random number.

Because of this, PRNGs don't prepopulate a long sequence of numbers for the computer to cycle through. Instead, they use a clever trick to achieve a similar result. That is, the PRNG only has to remember the pseudorandom number it generated previously, and then it *calculates the next number based on the previous number*.

In other words, there is indeed a prepopulated sequence. But we don't have to tell the computer the entire sequence *up front*. Instead, the computer figures out the next number based on the previous number.

Here's a simplified example to demonstrate the basic concept.

Let's make up a formula for generating random numbers between `0` and `210`. It'll go something like this:

1. Add `12549` to the previously generated random number.
2. Divide the result by `211`.
3. Grab the remainder. This remainder will be the next number in our sequence.

Yes, this sounds like a calculation that I pulled out of a hat, and that's because I did. But let's see what happens.

If the previous number generated by the computer was `73`, we'll apply our calculation:

```
(73 + 12549) % 211 = 173
```

Therefore, `173` will be the next pseudorandom number.

Now, the next time the computer generates a number after this, the computer will grab the previous number (`173`) and apply the same calculation to generate a new pseudorandom number:

```
(173 + 12549) % 211 = 62
```

And so on.

This type of PRNG is called a *Linear Congruential Generator*, or LCG. The name isn't terribly important, but you can impress your family and friends by saying that you spent your day creating your own Linear Congruential Generator.

There are a couple of other items I need to point out. You can't pull any old LCG formula out of your hat, like I admittedly did, as it takes mathematical expertise to choose the "best" formula. For example, the number we're dividing by (which in our example is `211`) must be a prime number. You'll read more about this soon.

There's also another loose end to tie up. We now have a method to generate the *next* random number in our sequence. But how do we determine the *first* number?

For this, the programming language provides a starting value, which is called the *seed*. The seed can be any arbitrary number hardcoded straight into the programming language and used the first time the computer is asked to generate a random number. From then on, the sequence continues based on the LCG's math formula. Many languages, including Python, use the current time to determine the seed.

A First Attempt at LCG

I whipped up the following code to serve as a basic LCG so that we can see what an LCG implementation looks like:

```
def generate_random_sequence(seed):
```

```
    sequence = []
    previous_number = seed

    while True:
        next_number_in_sequence = (previous_number + 12549) % 211
        if next_number_in_sequence in sequence:
            break
        else:
            sequence.append(next_number_in_sequence)
            previous_number = next_number_in_sequence

    return sequence
```

To run this code, I'll call the `generate_random_sequence` function and pass in an arbitrary seed of `999`:

```
print(generate_random_sequence(999))
```

This produces the following sequence:

```
[44, 144, 33, 133, 22, 122, 11, 111, 0, 100, 200, 89,
 189, 78, 178, 67, 167, 56, 156, 45, 145, 34, 134, 23,
 123, 12, 112, 1, 101, 201, 90, 190, 79, 179, 68, 168,
 57, 157, 46, 146, 35, 135, 24, 124, 13, 113, 2, 102,
 202, 91, 191, 80, 180, 69, 169, 58, 158, 47, 147, 36,
 136, 25, 125, 14, 114, 3, 103, 203, 92, 192, 81, 181,
 70, 170, 59, 159, 48, 148, 37, 137, 26, 126, 15, 115,
 4, 104, 204, 93, 193, 82, 182, 71, 171, 60, 160, 49,
 149, 38, 138, 27, 127, 16, 116, 5, 105, 205, 94, 194,
 83, 183, 72, 172, 61, 161, 50, 150, 39, 139, 28, 128,
 17, 117, 6, 106, 206, 95, 195, 84, 184, 73, 173, 62,
 162, 51, 151, 40, 140, 29, 129, 18, 118, 7, 107, 207,
 96, 196, 85, 185, 74, 174, 63, 163, 52, 152, 41, 141,
 30, 130, 19, 119, 8, 108, 208, 97, 197, 86, 186, 75,
 175, 64, 164, 53, 153, 42, 142, 31, 131, 20, 120, 9,
 109, 209, 98, 198, 87, 187, 76, 176, 65, 165, 54, 154,
 43, 143, 32, 132, 21, 121, 10, 110, 210, 99, 199, 88,
 188, 77, 177, 66, 166, 55, 155]
```

Conveniently, this sequence contains *all* the numbers from 0 to 210 with each number occurring once. It turns out that this sequence has great

uniformity and a period of 210.

However, this sequence is *not* so great when it comes to lacking pattern. Look at every pair of consecutive numbers: `44` and `144`, `33` and `133`, `22` and `122`. That's a pattern if I've ever seen one.

Short Circuit

Even without mathematical expertise, we can at least play with the numbers of the LCG formula to see if we can get a better pattern. If I change the formula to divide by `111`, which is not a prime number, we get this result:

```
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84,
90, 96, 102, 108, 3, 9, 15, 21, 27, 33, 39, 45, 51, 57,
63, 69, 75, 81, 87, 93, 99, 105, 0]
```

The first thing that jumps out at me is that this sequence has a short period.

In truth, this happens because when we get to the final number, `0`, and apply the formula to it, we get:

```
(0 + 12549) % 111 = 6
```

We get `6`, but you'll notice that `6` is the first number in the sequence, so the sequence starts over from the beginning. That is, once the "current" generated number is `6`, the next number *must* be `12` since that's what our LCG formula dictates. We're forced to return to the beginning of our sequence.

So, for an LCG to uniformly generate every number in a range of, say, `0` through `101`, the LCG must generate each of those numbers once and only once before beginning the sequence all over again. The sequence will "short circuit" as soon as we generate a number we've already generated before, and we are forced to jump back to that point in the sequence.

Now, if you were to ask Python to generate an integer between `0` and `9` using `random.randint(0, 9)`, it's possible to get the same result twice in a row. But then the sequence would get locked into generating that same number over and over again. If, for example, the LCG formula says that the previous result of `4` produces another `4`, then there's no way to ever stop generating a `4`!

The answer, though, is that when you execute `random.randint(0, 9)`, under the hood, Python is generating a much longer number using `random.random()`, such as `0.5339767180649452`, like you saw earlier. Python then does something along the lines of returning the final digit, like the `2` in our example.

However, when the computer generates the next number, it's performing the LCG formula on `0.5339767180649452`. This could produce a number like `0.11364654704325492`, whose final digit also happens to be a `2`. So it may appear as if you received the same number twice, but in truth, Python has generated a completely different number with many more digits. It's just that both numbers happen to end with a `2`.

This explanation might make sense for generating numbers between `0` and `9`, but what about generating numbers between, say, `116` and `865`? Indeed, the math that Python uses is slightly more complex than simply grabbing a final digit, but in any case, Python first generates a longer number and then converts it into a number in the range you're looking for.

A Better LCG Formula

You saw that our previous LCG formula of `(x + 12549) % 211` produced a sequence with a distinct pattern, which is problematic. Let's tweak our formula again, this time using the numbers: `(previous_number + 1223) % 227`.

This time, we get:

```
[179, 40, 128, 216, 77, 165, 26, 114, 202, 63, 151,
12, 100, 188, 49, 137, 225, 86, 174, 35, 123, 211, 72, 160,
21, 109, 197, 58, 146, 7, 95, 183, 44, 132, 220, 81, 169,
30, 118, 206, 67, 155, 16, 104, 192, 53, 141, 2,
90, 178, 39, 127, 215, 76, 164, 25, 113, 201, 62, 150,
11, 99, 187, 48, 136, 224, 85, 173, 34, 122, 210, 71, 159,
20, 108, 196, 57, 145, 6, 94, 182, 43, 131, 219, 80, 168,
29, 117, 205, 66, 154, 15, 103, 191, 52, 140, 1, 89, 177,
38, 126, 214, 75, 163, 24, 112, 200, 61, 149, 10, 98, 186,
47, 135, 223, 84, 172, 33, 121, 209, 70, 158, 19, 107, 195,
56, 144, 5, 93, 181, 42, 130, 218, 79, 167, 28, 116, 204,
65, 153, 14, 102, 190, 51, 139, 0, 88, 176, 37, 125, 213,
74, 162, 23, 111, 199, 60, 148, 9, 97, 185, 46, 134, 222,
83, 171, 32, 120, 208, 69, 157, 18, 106, 194, 55, 143, 4,
92, 180, 41, 129, 217, 78, 166, 27, 115, 203, 64, 152, 13,
101, 189, 50, 138, 226, 87, 175, 36, 124, 212, 73, 161, 22,
110, 198, 59, 147, 8, 96, 184, 45, 133, 221, 82, 170, 31,
119, 207, 68, 156, 17, 105, 193, 54, 142, 3, 91]
```

This has all numbers from 0 through 226. We have good uniformity, a longish period, and a pattern that is more difficult to detect. Now, there *is* a pattern (can you find it?), but since it's harder to see, our sequence has a greater appearance of randomness than before.

In the end, we managed to create a sequence of seemingly mixed-up numbers without having to store the entire sequence. Instead, we simply allow the LCG to do the work of computing the next number. The trick is to find the right formula, which takes mathematical expertise.

Luckily, mathematicians and computer scientists have developed a whole slew of PRNGs, some of which are LCGs and some that use other types of math. These PRNGs do the heavy lifting for us, and some have *awesome* names like Threefish, Philox, Mersenne Twister, and Blum Blum Shub. As of this writing, Python uses the Mersenne (pronounced mer-SEN) Twister, which is a type of LCG.

The Mersenne Twister is impressive, as it uniformly generates long numbers and also has a super-long period of about (2^{19937}). That's a

massive load of numbers before the sequence wraps around to the beginning again.

Another important attribute of the Mersenne Twister is that a computer can execute its calculations quickly. Some LCGs are so involved that it may take the computer a stretch of time to compute the next number. While the Mersenne Twister is itself a complex formula, it's one that a computer can execute with great speed. This is important, as you'll see in the next section that an operation like shuffling an array has to generate many random numbers in succession. If generating a single random number is not fast, shuffling an entire array can become way too slow.

TRNGs vs. PRNGs

Technology decisions are all about trade-offs, as I've mentioned, and both TRNGs and PRNGs have their pros and cons. The major advantage of TRNGs is that their numbers are much closer to being truly random; that is, the numbers are way less predictable. (Can *you* predict what the next lava lamp formation is going to look like?) PRNG numbers, on the other hand, are completely predictable once you know which formula is being used to fuel the PRNG.

Sometimes this difference is critical, as there are numerous applications where only truly random numbers will do. One of the most prominent examples is cryptography, which relies on the random generation of passwords and secret keys. For example, when we sign up for a new online service, our Internet browser asks us if it should generate a random password for us. If these passwords were only pseudorandom and generated using a predictable pattern, hackers would have a heyday breaking into people's accounts.

Another example is casino software. If a smart person could predict the combination of cards that the electronic poker machine will deal out next, the casino would go out of business fast. Even if something like Mersenne Twister is being used under the hood, a hacker may be able to find out the precise random number previously generated, and thereby be able to predict the next one.

For applications like these, PRNGs like Mersenne Twister aren't up to snuff. In the words of Python's documentation of the `random` module, "The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes."

On the other hand, for applications that don't absolutely need "true" random numbers, PRNGs have distinct advantages. First, PRNGs are practical; no radioactive material is needed. Secondly, PRNGs are generally faster, as a computer only needs to make an internal mathematical computation.

CSPRNG

It should be noted that there's also a type of random number generator called a Cryptographically Secure Pseudorandom Number Generator, or CSPRNG, which attempts to grab all the advantages of both PRNG and TRNG. That is, the computation is practical, fast, and cryptographically secure all at the same time. However, while CSPRNGs are indeed faster than TRNGs, they're generally not as fast as PRNGs. On the flip side, while CSPRNGs are much more secure than PRNGs, they're not as secure as TRNGs. So, it's a compromise that is appropriate for certain scenarios. A competent cybersecurity professional should be consulted if you're not sure whether you should be using a CSPRNG or a TRNG.

So much more can be said regarding generating random numbers, but it's time to get back to the topic of randomized algorithms. In particular, let's look at how to figure out the time complexity of shuffling an array.

The Fisher-Yates Shuffle

We've looked at several different sorting algorithms, but now let's turn our focus to an *un*sorting algorithm. While sorting makes order out of chaos, shuffling makes chaos out of order.

To shuffle an array, Python uses a variant of an algorithm known as the *Fisher-Yates Shuffle*, which was named after Ronald Fisher and Frank Yates, who first described it in 1938. This was before computers, so their original description involved using pencil and paper. Here's how it works as a computerized algorithm:

1. We point to the first index of the array. We'll eventually point to the others, too, but we start at the beginning. We'll call this the "current index."

2. We generate a random (or pseudorandom) number to choose a random index that is either the current index or *higher*.

   At the beginning of our algorithm, we can choose from all indexes. But if, say, our current index is 2, and the last index is 4, we'll choose a random index from 2 to 4, inclusive.

3. We swap the value at our current index with the value at our random index.

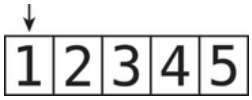
4. We move our pointer to the next index of the array.

5. We repeat Steps #1 through #4 until we reach the final value. Since there are no values to the right of the final value, swapping the final value with itself would be pointless, so the algorithm ends.

The Fisher-Yates Shuffle in Action

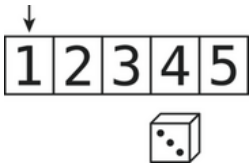
Let's visually walk through the Fisher-Yates Shuffle algorithm for the array `[1, 2, 3, 4, 5]`.

Step 1: We point to the first index in the array:



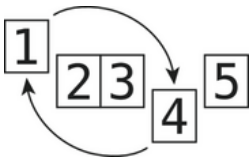
Step 2: We randomly choose one of the indexes from our pointer and rightward. For now, this can be any index of our array.

We'll indicate the random index with a die. Say that our randomly chosen number is a 3:

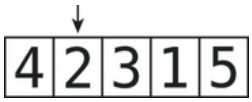


The number the die lands on represents the *index* that we're going to swap with. In this case, the index is `3` (and that slot happens to contain the value `4`). In the diagram, I've placed the die under its corresponding index.

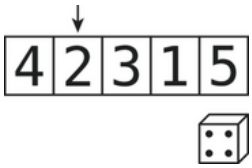
Step 3: We then swap the values of the current index (the arrow) and the random index (the die):



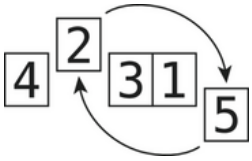
Step 4: We point to the next index of the array:



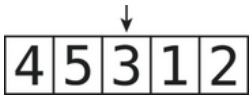
Step 5: We randomly choose a number between 1 and 4 inclusive since we're choosing an index from the current pointer and to the right. Say we roll a 4:



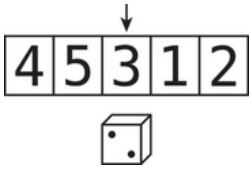
Step 6: We swap the values:



Step 7: We point to the next value:

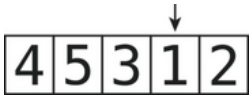


Step 8: We pick a random value from 2 to 4. Let's say that our computer chooses a 2:

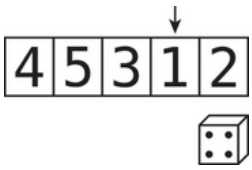


Step 9: In this case, we "swap" the value at index 2 with itself, leaving us with the same array as before.

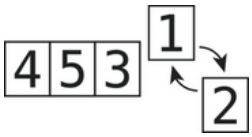
Step 10: We point to the next value:



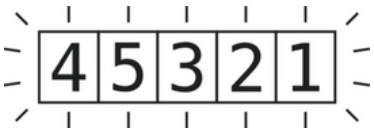
Step 11: We choose a random index. At this point, there are only two to choose from, 3 and 4. Let's say we roll a 4:



Step 12: We swap the current index's value with the random index's value:



There's no point in moving our pointer to the final index. Remember, according to this algorithm, we only swap a value with either itself or any value to its right. Since the final value can't be swapped with anything other than itself, we end our shuffle here:



Code Implementation: The Fisher-Yates Shuffle

Here's a Python implementation of the Fisher-Yates Shuffle:

```
import random


def fisher_yates(array):
    for i in range(0, len(array) - 1):
        j = random.randint(i, len(array) - 1)
        array[i], array[j] = array[j], array[i]

# Testing out our code:
array = [1, 2, 3, 4, 5, 6, 7, 8]
fisher_yates(array)
print(array)
```

Note that our loop stops before `len(array) - 1` since we aren't going to perform a swap for the final index.

The Efficiency of the Fisher-Yates Shuffle

Let's analyze the efficiency of the Fisher-Yates Shuffle. The steps of this algorithm consist of generating random numbers and performing swaps. For each of the N values in the array (besides the last one), we generate a random number and then perform a swap. All in all, this is N-1 number generations, and N-1 swaps, yielding 2N-2 steps, which boils down to O(N). This is way faster than any known sorting algorithm, and kind of makes sense since it's a lot easier to make chaos out of order than it is to make order out of chaos.

In terms of space, the Fisher-Yates Shuffle jumbles all the values in place and doesn't consume any extra memory.

Now that we've determined that shuffling takes O(N) time, we understand why shuffling an array before Quicksort is worth our while. As we've seen, if we suspect that we're dealing with a sorted array, Quicksort will have a speed of $O(N^2)$. But if we spend O(N) time preshuffling the array, we can reduce Quicksort's time to O(N log N), which makes preshuffling a potentially great move. That is, even with preshuffling, Quicksort's total time is still, from a Big O notation standpoint, O(N log N). This is because we have:

```
  N log N Quicksort steps
+ N preshuffling steps
```

which yields (N log N) + N, which is still O(N log N) since we drop the lower factor of "+ N".

So, preshuffling the array doesn't slow Quicksort down *at all* from a Big O perspective.

Shuffling the Wrong Way

You may be wondering why we need a special algorithm for shuffling. What's wrong with, say, the following alternative approach that I've invented off the top of my head? Here's my proposed algorithm:

Similar to the Fisher-Yates Shuffle, let's point to each of the array's values and swap them with other values. However, we won't constrain ourselves and only choose a value to the right of the pointer. Rather, we can swap each value with any other value—even those to the *left* of the pointer.

Here would be the code for this approach, which I'll call "naïve shuffling":

```
import random


def naive_shuffle(array):
    for i in range(0, len(array) - 1):
        j = random.randint(0, len(array) - 1)
        array[i], array[j] = array[j], array[i]

array = [1, 2, 3, 4, 5, 6, 7, 8]
naive_shuffle(array)
print(array)
```

Indeed, when I run the code, I get an array that looks shuffled. So, what exactly is wrong with naïve shuffling?

In truth, the problem is so subtle that it's terribly easy to overlook. But here's the thing: if we want an algorithm to be truly random, we must ensure that *every possible result of shuffling is equally likely to occur*.

For example, imagine that we have a six-sided die. But instead of having sides marked from 1 to 6, we have five sides marked 1, 2, 3, 4, and 5, and the sixth side is an extra 5. While the result of the die roll will certainly be random, it won't be perfectly random since the die is more likely to land on

a 5 than any other number. This hurts the uniformity of the possible results, thereby hampering the die's randomness.

We have to apply the same analysis to shuffling. If we're shuffling the array `[1, 2, 3]`, there are six possible permutations we can end up with:

```
[1, 2, 3] [1, 3, 2]
[2, 1, 3] [2, 3, 1]
[3, 1, 2] [3, 2, 1]
```

Just like the die, we want to make sure that each of these six possibilities is equally likely to result from our shuffling algorithm. We don't want a situation where one of these permutations is more likely to occur than the others.

With that in mind, let's now analyze our proposed naïve shuffling algorithm using the same example array of `[1, 2, 3]`.

With naïve shuffling, we point to each value in the array and "roll a die" by choosing a random index from *anywhere* in the array to swap it with. In our example, this is like rolling a three-sided die and getting the possible results of `0`, `1`, or `2`. This will result in combinations such as `1, 2, 0` or `2, 2, 1` or `0`, `0`, `0`. There are, in fact, 27 possible sets of random numbers that may be chosen. Here goes:

```
0, 0, 0 --- 1, 1, 1 --- 2, 2, 2
1, 0, 0 --- 0, 1, 0 --- 0, 0, 1
2, 0, 0 --- 0, 2, 0 --- 0, 0, 2
1, 1, 0 --- 1, 0, 1 --- 0, 1, 1
1, 1, 2 --- 1, 2, 1 --- 2, 1, 1
2, 2, 0 --- 2, 0, 2 --- 0, 2, 2
2, 2, 1 --- 2, 1, 2 --- 1, 2, 2
0, 1, 2 --- 0, 2, 1 --- 1, 0, 2
1, 2, 0 --- 2, 1, 0 --- 2, 0, 1
```

To be clear, these aren't the permutations of the results of the shuffle itself. These are the possible combinations of *random indexes* (die rolls) that we'll

generate throughout our shuffle.

Now, if we take each of these permutations and compute the result of the shuffle for each permutation, we get:

```
[3, 1, 2] --- [2, 3, 1] --- [3, 1, 2]
[3, 2, 1] --- [3, 2, 1] --- [2, 3, 1]
[1, 3, 2] --- [2, 3, 1] --- [2, 1, 3]
[3, 1, 2] --- [1, 3, 2] --- [1, 3, 2]
[2, 1, 3] --- [2, 1, 3] --- [3, 1, 2]
[2, 1, 3] --- [2, 3, 1] --- [1, 3, 2]
[3, 2, 1] --- [3, 2, 1] --- [2, 3, 1]
[1, 2, 3] --- [1, 2, 3] --- [1, 2, 3]
[1, 3, 2] --- [1, 2, 3] --- [2, 1, 3]
```

If we take a tally of how many times each of the shuffling results occurs, we get:

```
[1, 2, 3] -> 4 times
[1, 3, 2] -> 5 times
[2, 1, 3] -> 5 times
[2, 3, 1] -> 5 times
[3, 1, 2] -> 4 times
[3, 2, 1] -> 4 times
```

Aha! It turns out that our naïve shuffling algorithm will produce three of the permutations more often than the other three permutations. This means that our algorithm doesn't produce uniformly random results.

The Fisher-Yates Shuffle, on the other hand, does, and here's why.

For an array of size 3, with Fisher-Yates, we only generate *two* random numbers. Again, this is because we don't bother to generate a random number when we're up to the final value in the array. As we tally all the possible permutations of random numbers, we should also keep in mind that the range of random numbers decreases as we progress through the array. For example, when we're at the first value, we'll generate a random number

between 0 and 2. But when we're pointing to the second value, the algorithm will only generate a number between 1 and 2.

Based on this, here are all the permutations of die rolls we might get in our example of shuffling an array of size 3:

```
0, 1
0, 2
1, 1
1, 2
2, 1
2, 2
```

We have only six permutations. When we compute the results of shuffling based on the die rolls, we get:

```
[1, 2, 3] -> 1 time
[1, 3, 2] -> 1 time
[2, 1, 3] -> 1 time
[2, 3, 1] -> 1 time
[3, 2, 1] -> 1 time
[3, 1, 2] -> 1 time
```

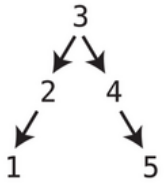
We end up getting exactly one instance of each possible permutation of the shuffled array, which means that Fisher-Yates produces random results with perfect uniformity. And that's why Fisher-Yates is the gold standard when it comes to shuffling. It's not so intuitive at first, but once we analyze all the possible permutations, it's easier to see the difference between Fisher-Yates and our proposed naïve shuffling.

Binary Search Tree Randomization

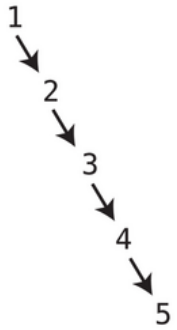
So far, we've looked at various types of randomized algorithms. Generating a random number is perhaps the "ultimate" randomized algorithm, as it is what powers all of the other randomized algorithms. Shuffling an array is also a randomized algorithm since it uses randomization to jumble the order of the array's values.

Randomized Quicksort, another randomized algorithm we've seen, uses randomization for the sake of creating order efficiently. Although the algorithm's goal is to create order rather than randomness, it is still considered a randomized algorithm since it utilizes randomization somewhere within the algorithm's steps. In fact, this utilization of randomization can be useful in any scenario where having sorted data would be to our disadvantage. Another example of this pops up with regard to the building of binary search trees.

I noted back in Volume 1, Chapter 15, that the order of our data could significantly affect the formation of our tree. For example, if we want to build a tree out of the data `[3, 2, 4, 1, 5]` by pulling out each value from left to right and inserting them into the tree, we get this nicely balanced tree:



However, if those same values were in the order `[1, 2, 3, 4, 5]`, our tree would come out like this:



This tree is essentially nothing more than a linked list and loses all the efficiency advantages that a binary search tree has over a linked list. So, this is another instance where sorted data works to our disadvantage.

However, if we have reason to suspect that our data might be in sorted order, we can do our little trick of preshuffling the array before building the tree! The shuffling of an array will greatly reduce the odds that the array is in sorted order.

Randomization for Distribution

Another interesting use of randomization is for distributing items evenly. This is another example of where randomization is counterintuitively used for the sake of creating order.

Probability theorists like to discuss the concept of putting "balls into bins." That is, say we have 10 bins (or boxes, if you like) and want to distribute a whole bunch of balls into them evenly. This is easy to do if we know how many balls we have. If we have 1,000 balls, we place 100 balls in each bin. But what if we don't know how many balls we have? How can we ensure that we distribute them evenly?

In such cases, we can still fill the bins evenly if we use a "round-robin" approach. That is, we rotate through the bins, putting a single ball in each bin. As long as we follow this pattern consistently, the bins will be filled practically evenly.

Now, here's a question to ponder: what happens if we take each ball, one at a time, and place it in a *random* bin?

Here's some code that does this. Here, we treat arrays as "bins" and integers as "balls." That is, we throw 1,000 integers into 10 different arrays, choosing a random array for each integer:

```
import random


bins = [[], [], [], [], [], [], [], [], [], []]

for ball in range(1000):
    bin = bins[random.randint(0, 9)]
    bin.append(ball)

for bin in bins:
    print(len(bin))
```

Because we choose the bins randomly, each time I run this code, I get a slightly different result. However, all the results are similar. Here's one outcome of how many balls each bin contains:

```
103
95
101
105
88
91
105
113
106
93
```

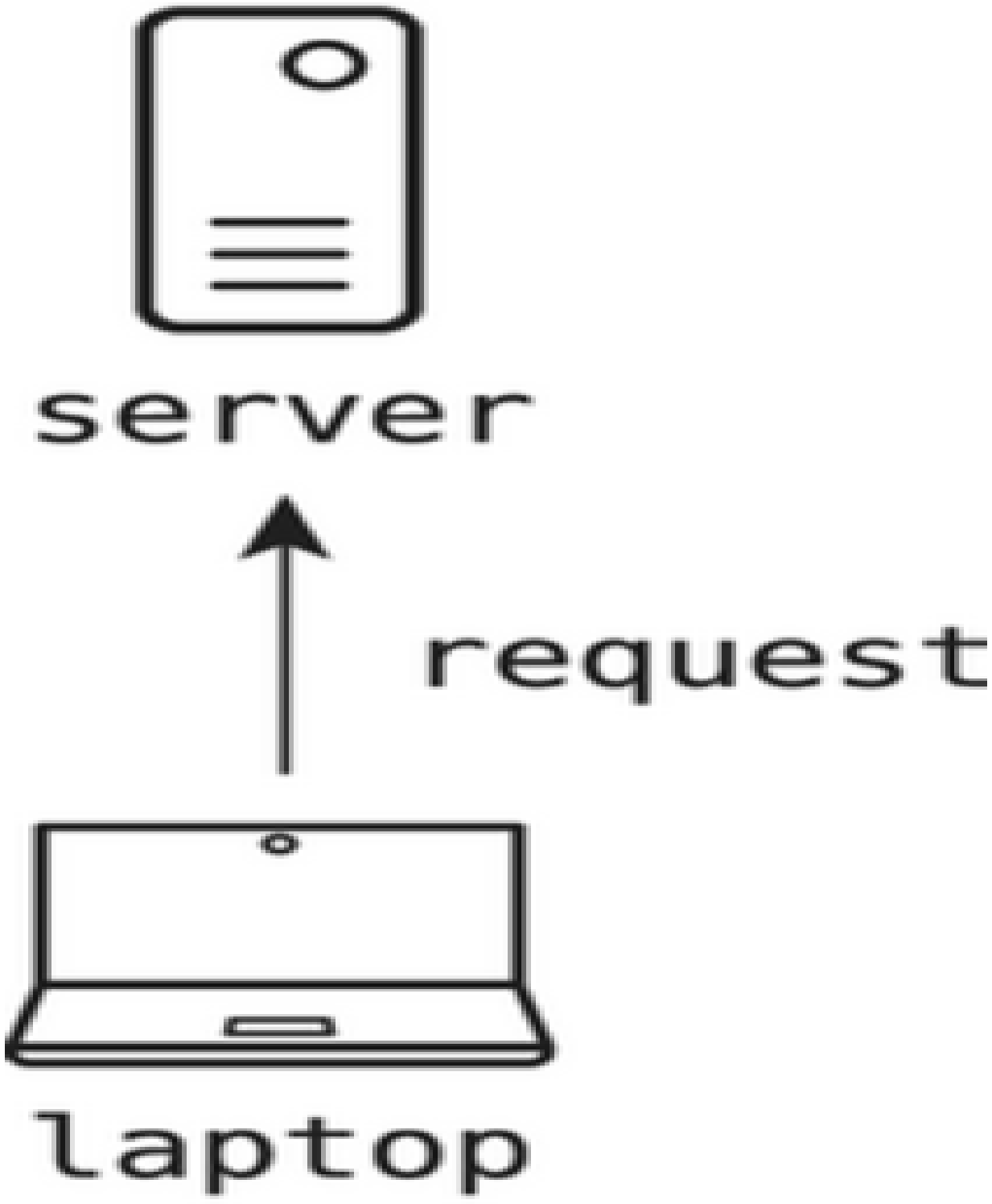
Whoa. While the balls haven't been distributed *perfectly* evenly, it's surprisingly close.

Because each bin is chosen at random, and each bin is as likely to be chosen as every other bin, the laws of probability dictate that the bins are likely to contain roughly the same number of balls. Probability theorists use math to define how likely this is, but for our purposes, it's likely enough. This may not be true with a small number of balls relative to the bins, but when we have many more balls than bins, the distribution is pretty uniform. Feel free to play with the code and change the number of "balls"—it's interesting to see the results.

Load Balancing

Now, you didn't buy this book to learn about throwing balls into bins. However, this concept is used in various distribution algorithms. One common use case is *load balancing*. Load balancing is a concept commonly used for Internet servers and similar networks. If you're not familiar with load balancing, here's the gist of it, using Internet usage as a basic example.

When you open your Internet browser and visit a website, your browser makes a request to a server to receive the web page content, as illustrated in the following figure:

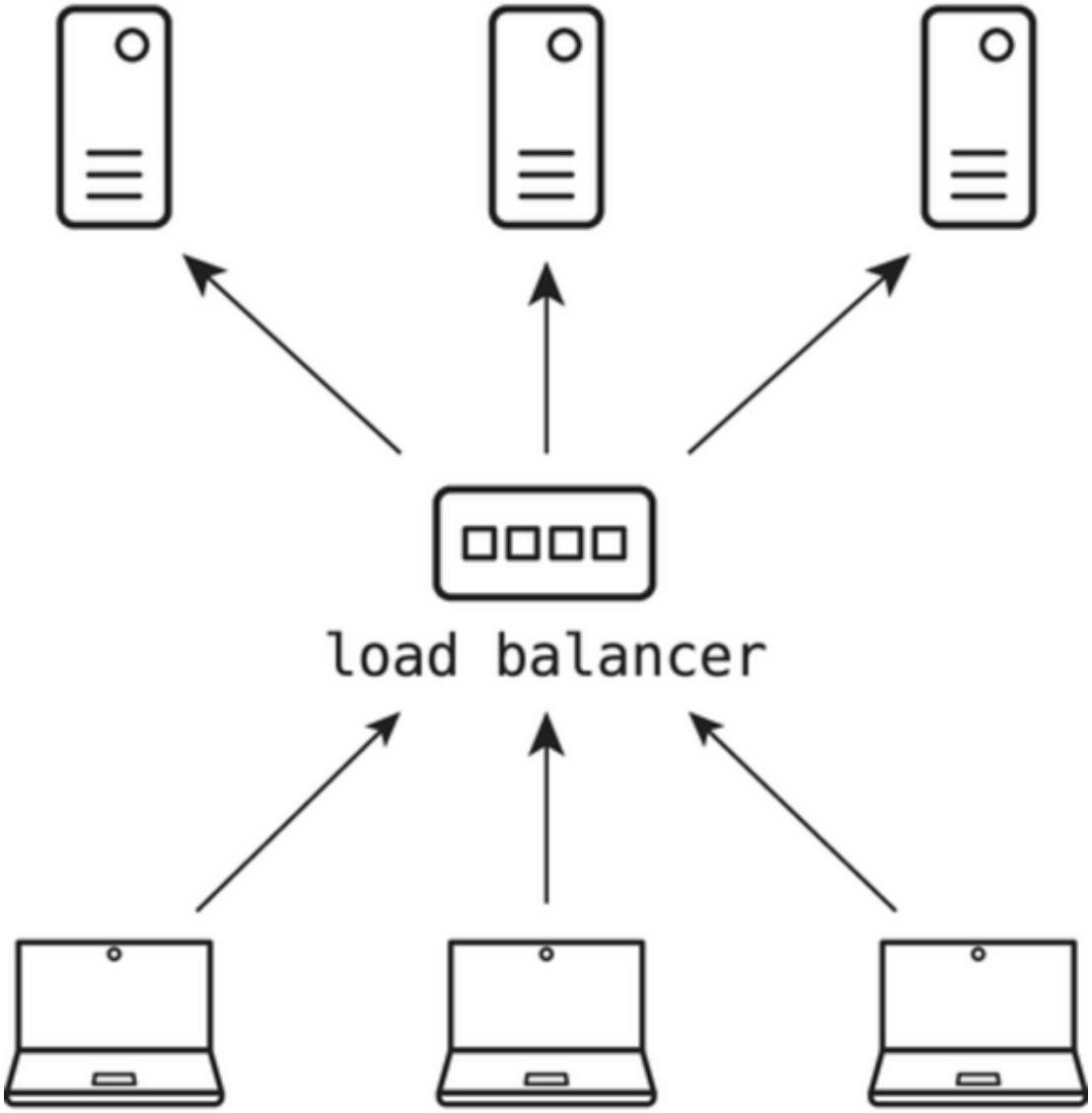

server
request
laptop

A server is simply a computer dedicated to delivering content to your computer; it “serves” the content to you.

Now, a server can only handle a finite amount of traffic at a given moment. If a website is so popular that thousands of people visit it each second, a single server may not be enough to deliver content to everyone in a timely manner. Because of this, many websites are delivered using a system made up of *multiple* servers. Each server is capable of delivering the same content, so no matter which server your laptop talks to, it will receive the same web page.

However, a laptop doesn’t choose which server it wants to make a request from. If it did, and one million users of the website decided to all make requests from the same server, we haven’t accomplished anything by adding multiple servers.

To make this setup work, a device known as a *load balancer* is used to direct all traffic. That is, each laptop makes a request of the load balancer, and the load balancer routes the request to one of the servers, like so:



Even though this example system has a bottleneck of having only one load balancer, it’s still a win. This is because a load balancer can generally act much faster than a server since all it does is route requests, as opposed to the server that interacts with databases and performs more time-intensive tasks. So, even though all the traffic may be flowing to that one load balancer, the load balancer may still be able to direct all the requests quickly enough.

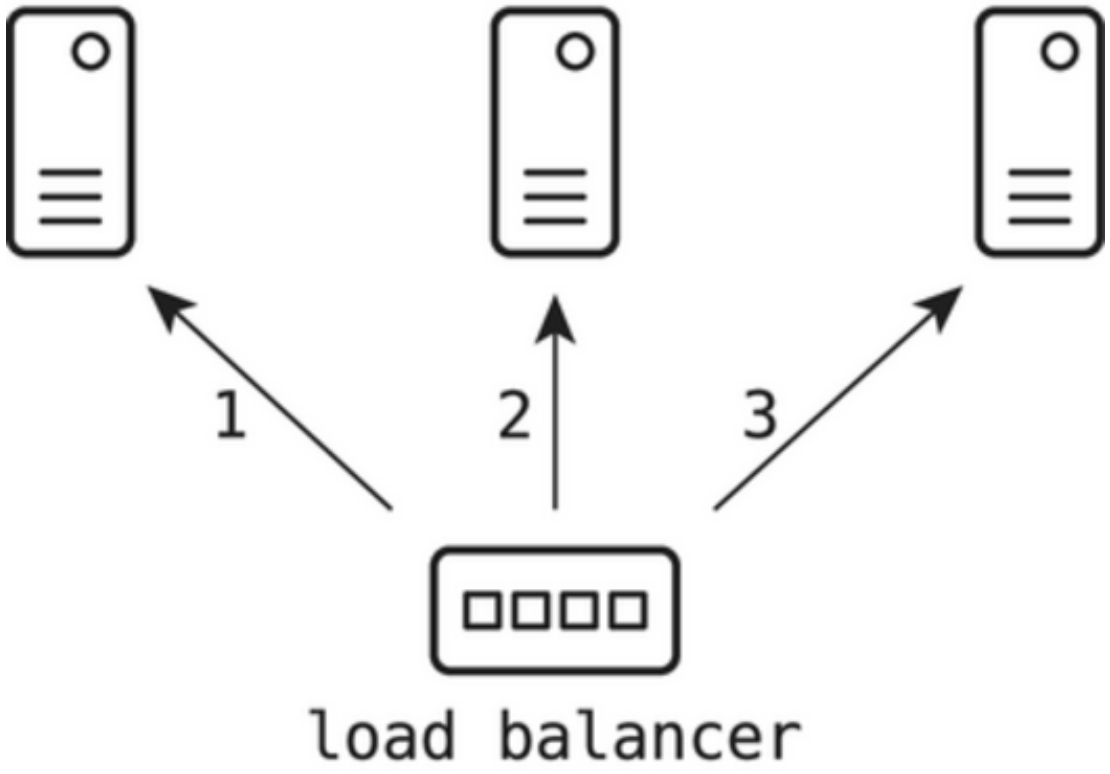
Now, here's the crucial thing. For this setup to be effective, the load balancer needs to distribute the traffic *uniformly* among all the available servers. Certainly, if the load balancer sends all the traffic to one server, we've truly accomplished nothing. We want all the servers to take on an equal share of the load.

And this is where the balls-in-bins analogy becomes relevant because directing requests to servers is essentially the same concept.

Fortunately, there are numerous algorithms that load balancers can use to ensure that the traffic is distributed evenly.

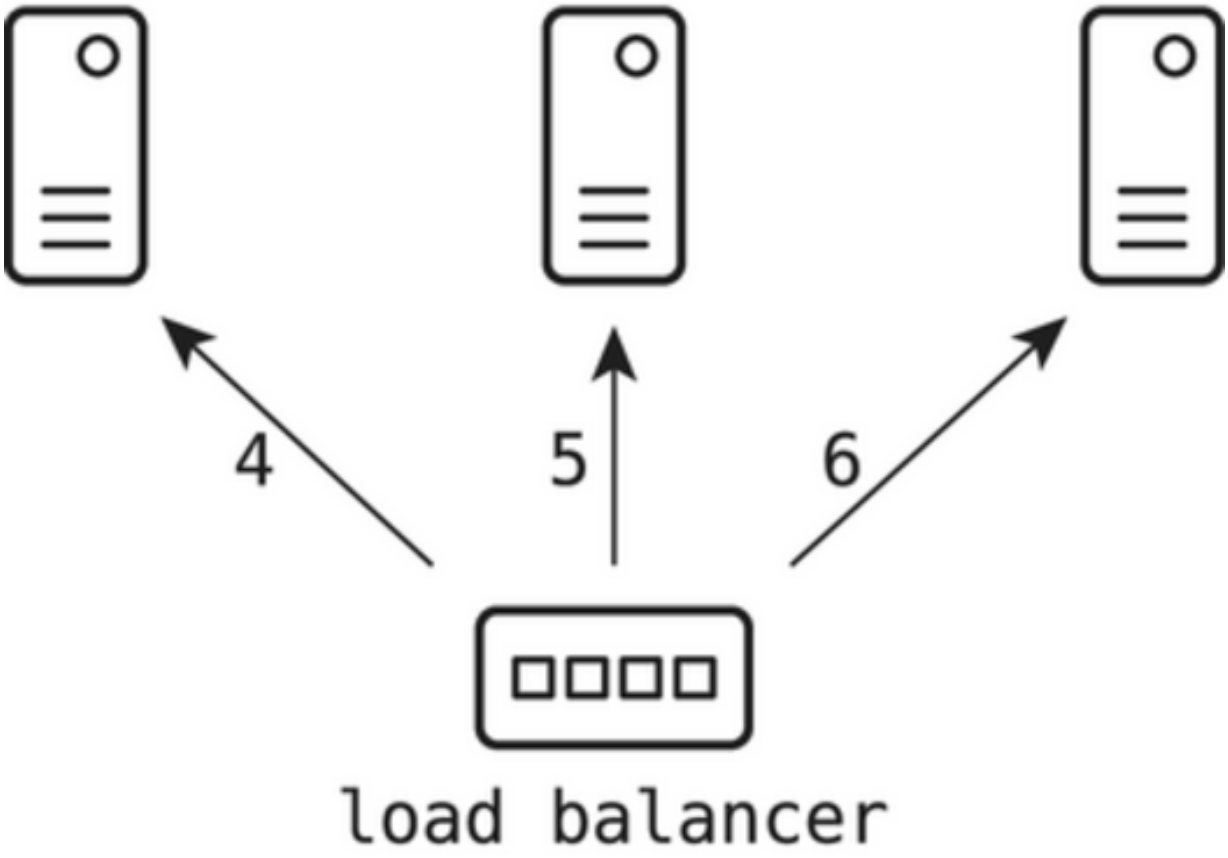
Round-Robin Load Balancing

One of the simplest and most common load balancing algorithms is called *round-robin* load balancing. We rotate between all the servers and send each request to the next server as illustrated in the figure.



In this image, we send the first request to the first server, the second request to the next server, and the third request to the next server.

With subsequent requests, we repeat the same pattern as shown here:



And so on. As with balls-in-bins, round-robin load balancing distributes the requests perfectly evenly, and the algorithm is as easy as pie.

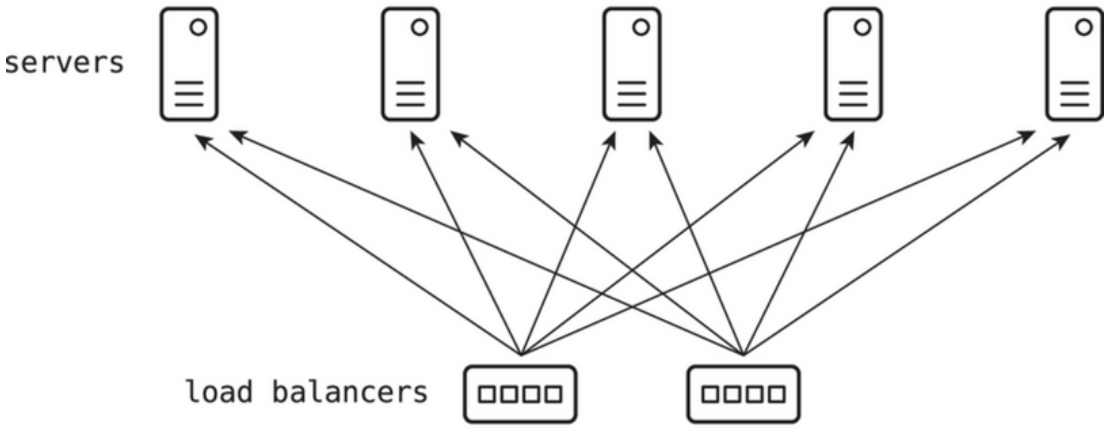
However, the round-robin approach isn't always the best one. This is because servers don't always deliver responses at the same speed. This can happen for a variety of reasons, among them that one user made a request that is complex and requires that request's server to do a lot more work.

Least-Time Load Balancing

A potentially "smarter" load balancing algorithm, called *least-time load balancing*, is one in which the load balancer *detects* how quickly each server is currently taking to process requests. Once the load balancer is

aware of how speedy each server is in the moment, the load balancer sends the next request to the fastest server.

However, this works best when we're only dealing with a single load balancer. Some setups, though, include *multiple* load balancers, which can be necessary when we have *a lot* of traffic, as shown in the figure.



But here's the gotcha when dealing with multiple load balancers. In that diagram, we have two load balancers, but imagine we have 10.

Let's also say that the 10 load balancers all receive requests at the same time. With the least-time load balancing scheme, the load balancers will all find the fastest server, and *send all 10 requests to it*. What a moment ago was the fastest server may now become the slowest by far, as we've clogged it with a slew of 10 requests simultaneously!

So, least-time balancing can fall apart when dealing with multiple load balancers. Luckily, there's yet another load balancing algorithm that works well even under these conditions, and it involves randomization.

Randomized Load Balancing with the Power of Two Choices

Indeed, we can use randomization as the basis of a load-balancing algorithm. The simplest way to do this is to have the load balancer send each request to a *random* server. As you saw with balls in bins, distributing

balls into random buckets is an effective way to distribute the balls pretty evenly. This distribution won't be as uniform as round-robin load balancing, but it'll be uniform enough. We'll call this approach the "purely random" load-balancing algorithm. However, purely randomized load balancing runs into the same pitfalls as the round-robin approach: if there's one slow server, we're going to keep sending requests to it even though we should be avoiding it until it speeds up again.

However, a randomization algorithm known as *the power of two choices* works astonishingly well. This approach is almost as simple as the purely random algorithm, but with a twist: each load balancer chooses *two random* servers, checks which one is faster in the moment, and sends the request to that faster server.

Again, the problem with the purely random approach was that we may end up sending requests to the slowest server. But with the power of two choices, this is impossible to do. Even if our randomly chosen servers happen to be the slowest server and the second-slowest server, we still avoid sending the request to the slowest server.

The power of two choices is beneficial even if we randomly pick the fastest and second-fastest servers. After all, we'll shave off some extra time by sending the request to the fastest server instead of the second-to-fastest one.

In theory, we could also have a power of *three* choices algorithm, which randomly chooses three servers and sends the request to the fastest one. However, the more servers we choose from, the more likely it is that we have multiple load balancers sending requests to the same server at once. Ultimately, these numbers may vary depending on the number of load balancers and servers our system has. However, as a general rule, the power of *two* choices algorithm has held up in many cases and has been shown to be effective.

Power of Two Choices for Balls in Bins

To see the power of two choices in action, let's go back to our balls-in-bins code. Let's change it up so that instead of throwing each ball into a random bin, we choose *two* random bins, and throw the ball into the *emptier* bin. This is analogous to a load balancer sending a request to the faster of two servers. Just as we want each server to have an equal load of web requests, we want each bin to contain the same number of balls. Here's our updated code:

```
import random


bins = [[], [], [], [], [], [], [], [], [], []]

for ball in range(1000):
    bin_1 = bins[random.randint(0, 9)]
    bin_2 = bins[random.randint(0, 9)]

    if len(bin_1) < len(bin_2):
        bin_1.append(ball)
    else:
        bin_2.append(ball)

for bin in bins:
    print(len(bin))
```

No matter how many times I run this code, I get results along these lines:

```
100
99
100
100
100
101
100
99
100
101
```

Wow! That's pretty close to perfect distribution. And that, my friends, is the power of two choices—and randomization.

The Probability Theory Caveat

Some of the randomization concepts we've looked at are intuitive, while others are less so. It's important to remember that probability and statistics are an entire subject, nay, an entire *field*, and I can't do justice to it in these pages.

With that in mind, if you are developing your *own* randomization algorithm, you'll often need to check its effectiveness using established probability theory. Sometimes it may *seem* that randomization will produce fantastic results, but when you run your idea through statistical formulas, you may discover that it's not so hot after all. Remember when I thought that naïve shuffling was a good idea?

That being said, benchmarking is also your friend, and you may be able to demonstrate the power of your algorithm using the techniques outlined in the previous chapter.

Wrapping Up

In this chapter, you learned what randomization algorithms are and how they can optimize various algorithms, such as Quicksort and building binary search trees. You looked at the fundamentals of how computers generate random numbers and the right way to use random numbers to shuffle an array. You also discovered how the power of two choices uses randomization to achieve an ideal load-balancing algorithm.

But even with all this, we've only begun to scratch the surface of randomization algorithms. In fact, randomization algorithms will play an important role throughout many of the remaining chapters of this book.

In the next chapter, we're going to explore a fundamental set of data structures and algorithms surrounding the idea of *caching*, which can drastically improve the speed of your code. And we'll extend the ideas of this chapter to explore how randomization can be used to optimize caching.

Exercises

The following exercises provide you with the opportunity to practice with randomization algorithms. The solutions to these exercises are found in the section *Chapter 3*.

1. Write a function that *randomly* chooses 3 different values from an array, and returns a brand-new array that contains those 3 values. Ensure that the values appear in the same order as they appeared in the original array.

   For example, if the original array is `[7, 1, 5, 2, 9, 0, 3, 6, 4]`, and the computer selects the `9`, `6`, and `5` (in that order), we should return `[5, 9, 6]` since that's the order in which those three integers appear in the original array.

2. Write a function that randomly chooses a single key from a hash table (in other words, a dictionary, if you're using Python).

3. *Puzzle:* There are several simple ways to randomly select a single item from an array, such as using Python's built-in `random.choice` method, for example. Another basic approach is to use `random.randint` to randomly select one index from the array, and return the value at that index.

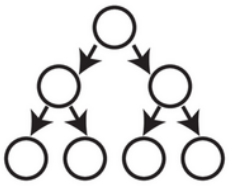
   However, there's a clever but more involved approach that can be handy in certain scenarios. I'm only going to describe *part* of the algorithm, and your challenge is to fill in the rest. The algorithm goes like this:

   We run a loop that iterates over each value in the array. Within each iteration of the loop, we perform a *certain computation* (which I will not reveal here) to decide whether the value we're currently pointing to should be the value we're selecting. If the computation decides to

select the current value, we return that value and we're done. If the computation decides to *not* select the current value, we simply continue with the next iteration of the loop. If the loop reaches the final item of the array, then that final item becomes the value we're choosing.

Your job is to figure out what this certain computation is. The tricky (but tantalizing!) part is to figure out how to ensure that each item has an equal chance of being selected.

4. *Puzzle:* Devise an algorithm that chooses a random node from a binary search tree. You can assume that the tree is complete (that is, a tree whose levels are entirely full), like this one:



To make this exercise more challenging, make sure that the function runs in O(log N) time and doesn't consume any extra space.

Chapter 4

Cache Is King

In the previous chapter, I introduced randomized algorithms and explained how randomization can boost the efficiency of all sorts of applications. In this chapter, you'll discover how randomization can also boost the speed of a process known as *caching*.

Caching is built into the hardware of virtually all computers and affects the speed of the code you write. At the same time, you can create software that performs additional layers of caching as well.

But whether you write your own caching code or leverage the caching built into your computer, understanding caching—and how randomization can play a role in it—will allow you to take your code's speed to the next level.

In addition, caching is another key piece of the puzzle of why two algorithms can have different actual-time speeds even though they execute the same number of steps. Understanding the related concept of *spatial locality* can help you design faster algorithms even when Big O analysis tells you that you can't beat the speed of the algorithm you already have.

So, without further ado, let's dig into caching.

Caching

A fundamental rule about computers is that the farther the data is from your computer, the longer it takes for your computer to retrieve it. For example, you can open a file that's local to your computer more quickly than you can download information from the Internet. This is because the Internet consists of servers that are *outside* your computer, and it takes more time for your computer to obtain those servers' data.

Let's take a look at an example of how this can be a major deal for the software we write.

Imagine that we're building an app that searches the web for the cheapest price available for various physical products. The user enters something like "Vroom-Master Vacuum Cleaner 3000," and our app scours the Internet to find whichever online retailer is selling it for the cheapest price. Note that we're not building a massive database that stores a gazillion products and their prices. Instead, our software is searching the web each time the user searches for a particular product. In a sense, the web is our "database."

Now, say that the Vroom-Master's latest model is becoming a hot fad; everyone you know is buying one. Suddenly, our app finds itself repeatedly searching the web for the best bargain for the Vroom-Master 3000. Assuming that stores are not constantly changing their prices at a rapid clip, it's kind of a shame that our software has to search the web *each and every time* someone asks for the Vroom-Master 3000. Searching the web takes time! Wouldn't it be nice if the app could *remember* information the first time it finds it and then not have to search the web again and again for the same information?

Luckily for us, that's exactly what a cache does.

What Is a Cache?

A *cache* (pronounced "cash") is simply a data container that takes data that was retrieved from a faraway source and stores it locally. (The word "cache" can also be used as a verb. That is, our computer can *cache* data it gets from the web.) Because the data is more local, we'll be able to retrieve it more quickly in the future. The definition of "local" can change based on context, but for now, let's say that data that lives on our computer is considered local, while data retrieved from the Internet is "far away."

In theory, we can create our own code-based cache for the app we're building. That is, our app will take data it gets from the web and store it on the computer, tablet, or smartphone that the software is operating on. We may, for example, have our code initialize some data structure and store the data in it.

So, the first time someone searches for the Vroom-Master 3000, our app will search the web for it and then save the desired information in the user's local data structure, which serves as our cache. This way, the next time someone asks for this information, our app doesn't have to search the web again; it can instead retrieve the data from the cache.

Now, this behavior may not be desirable for websites that are constantly changing. In this case, the cache can become what is known as *stale*. That is, the website may have been updated with new information, but the app is pulling up the outdated info saved in the cache. But for websites that don't update often, the local cache can save us a lot of time.

Code Implementation: A Hash-Table Cache

We have a variety of options as to which data structure we might choose to serve as our cache. However, a hash table is a natural fit for housing a cache since data can be stored and retrieved from a hash table in O(1) time. The following sample code demonstrates how we might use a hash table to serve as a cache:

```
import time
```

```
cache = {}

def lowest_price(product):
    if product in cache:
        return cache.get(product)
    else:
        return search_web_for(product)

def search_web_for(product):
    # Actual web-searching code goes here. Since we're not
    # actually going to search the web, we'll just use mock
    # data about the price and online shop:
    data_from_web = [799, "Jupiter Electronics"]
    # To mimic the time it takes to search the web, we'll pause for
    # one quarter of a second:
    time.sleep(0.25)
    # Cache the retrieved data:
    cache[product] = data_from_web

    return data_from_web
```

Here, the main function is `lowest_price`, which tells the app to first check the `cache` to see if it already contains data for that product. Only if the `cache` does *not* contain that data does the app fetch the data from the web.

In the `search_web_for` function, the app mimics searching the Internet by sleeping for a quarter of a second. In addition, the code uses the mock data `[799, "Jupiter Electronics"]` to indicate the product's lowest price and the online shop where the product is sold for that price. In this example, the product costs $799 and can be found at that price at a store called Jupiter Electronics.

Now, suppose that our software searches for products using the following commands:

```
print(lowest_price("Vroom-Master 3000"))
print(lowest_price("Dustpan Deluxe"))
```

```
print(lowest_price("Vroom-Master 3000"))
print(lowest_price("Vroom-Master 3000"))
print(lowest_price("Dustpan Deluxe"))
print(lowest_price("Vroom-Master 3000"))
```

The first time we search for `"Vroom-Master 3000"` and `"Dustpan Deluxe"`, it'll take a quarter of a second to obtain the data for each. However, in all subsequent requests, we'll pull the data from the cache, enabling these requests to occur much more quickly.

Out of Space

Now that you know how awesome caching is, I could end the chapter here. However, there's one itty-bitty teeny-tiny little problem. If our cache keeps saving more and more information from the outside world, our computer (or tablet or smartphone) is going to run out of space quickly. After all, we can't expect our cache to store the entire Internet!

To manage these space constraints, we need to ensure that our cache is not storing *all* data we've retrieved from the web. To do this, at some point we'll have to remove some data from our cache—or at least prevent new data from entering. This, then, is the tricky part of caching. Somehow, we need to figure out what information we want to keep and what information to get rid of.

Eviction Policies

Here's a bit of cache jargon I'm going to use going forward. The common term for deleting something from the cache is to *evict* it. (It sounds harsh, I know.) Similarly, an *eviction policy* is an algorithm that decides what data we should evict from the cache.

Computer scientists have proposed a whole slew of different eviction policies over the years. However, the trick is to find the most *efficient* eviction policy for our cache. To define what it means for an eviction policy to be efficient, let me first introduce a few more caching terms.

Recall that each time an application makes a request for data, it first checks to see if the data is in the cache. If it is, this is called a *cache hit*. Cache hits are good since it means that the app can grab the data from the cache rather than fetch the data from an external source. On the other hand, if an app makes a request and the data is *not* in the cache, this is known as a *cache miss*. Whenever there is a cache miss, the app has no choice but to look for the data on the Internet or wherever the external data source is.

We can use these terms to help define what it means to have an efficient eviction policy. That is, an efficient eviction policy works to *increase cache hits and decrease cache misses*. (We'll soon see that we accomplish this by trying to only store data that we'll need again in the future.) The more requests that our app can fulfill by getting data from the cache, the less time our app needs to spend searching the web.

In sum, the more efficient our cache is, the faster our software will run.

Farthest-in-Future Eviction Policy

Let's think about how we might design an efficient eviction policy.

Ideally, we'd evict data that will never be requested ever again. After all, the whole point of caching is to quickly deliver data on the second and third time it's requested. So, if there will never be another future request for a particular item, we can safely evict its data. This enables us to save room for data that we *will* request again at some point. (How will we know what data will or won't be requested again? Good question, but hold onto it for now.)

Now, even if all the products in our sample app will be requested again at some point, we can still make our eviction policy more efficient by evicting data that will be requested *farthest into the future*. Let's see an example of what I mean.

Here's a cache that stores up to four items:

Right now, the cache is empty, which, incidentally, is called a *cold cache*. (I like calling it a cold *hard* cache, but no one else calls it that.)

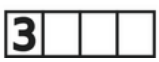
To keep the diagrams nice and small, instead of requesting items such as a "Vroom-Master," we'll be requesting integers. So, imagine in your mind's eye that each integer represents some particular physical product.

Here's an example sequence of requests we'll be making, from left to right:

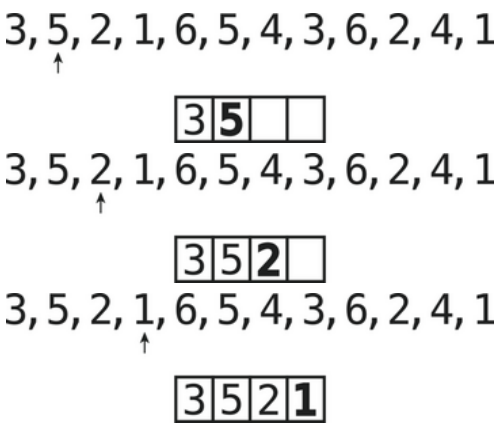
3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1

Let's begin to make our requests. First, we request the 3. There's plenty of room in the cache, so we cache the 3:

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1



We do the same for the next three requests, filling up the cache:

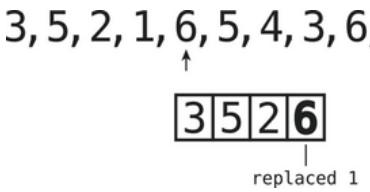


Next, we request the 6. The 6 is not currently in the cache, so this is a cache miss. The previous requests were also cache misses, but this is the first cache miss we encounter where the cache is full.

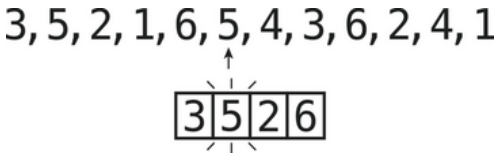
Because our cache is full, we need to evict one of the cache's current items if we're going to cache the 6. And so, we look to our eviction policy to decide which item we'll evict.

If we look ahead to our future requests (those to the right of the 6), we'll see that of all the items currently in our cache, the 1 is the one we'll be requesting farthest into the future. That is, the 3, 5, and 2 will all be requested sooner than the 1. This is what I mean by the term *farthest-in-future* eviction policy; we evict whichever item will be requested later than all other items in the cache.

In this case, we'll evict the 1 and replace it with the 6:



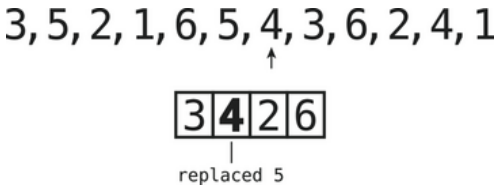
Next up, a 5 is requested. We happen to have a 5 in the cache, so we have our first cache hit!



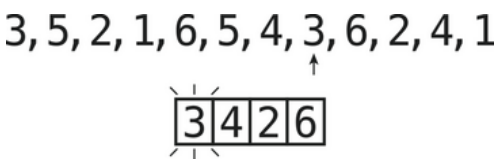
We saved time from having to request the 5 from the web. Also, we don't have to evict anything since we're not inserting any new data into the cache.

Next, a 4 is requested. This is a cache miss, but what item to evict?

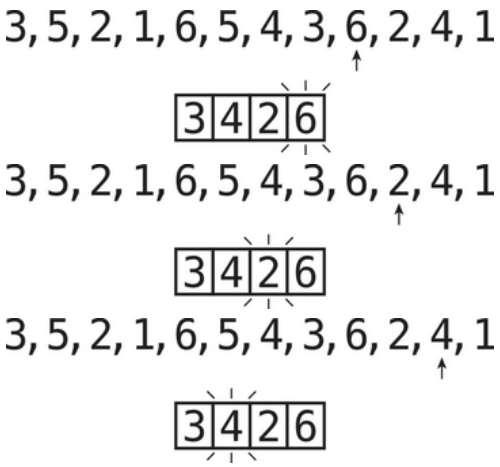
The 3, 2, and 6 will all be requested again soon, but there won't be a request for 5 again in the near future. Perhaps there will be in a future batch of requests, but there aren't any 5's in the current list of requests. So let's evict the 5 and cache the 4:



Next, we request a 3. That's a cache hit:



We also have cache hits with the next three requests:



The final request in this sequence is a 1. That's a cache miss. As to which item to evict, it's hard to say since we're not yet privy to whatever requests may come next.

In any case, the walk-through shows what it means to evict items that will be requested farthest into the future. But now let me explain why this eviction policy is ideal. To some, this may already be intuitive, but I'll spell it out anyway.

If we evict items that are about to be requested again soon, we'll certainly cause cache misses that could have been avoided. Now, one might argue that perhaps there might be a case where it's worthwhile keeping a farthest-in-future item if that item will be requested many times down the line. However, this is not a valid concern, as the next time that item is requested, it'll be cached at that point and available for all those subsequent requests. By keeping it around in the meantime, we're taking away space from items that are being requested sooner and causing cache misses.

Clairvoyance Is the Best Policy

The entire premise of our proposed eviction policy assumes that *we can predict* what requests will be made in the future. But this raises the question: are we able to see into the future?

Indeed, some computer scientists refer to farthest-in-future policy as the *clairvoyant* eviction policy. That is, we can only evict farthest-in-future items if the computer can, somehow, see the future and know what requests will occur down the road and when. But again, this isn't practical in most cases because computers *cannot* see into the future (yet).

That being said, there are some applications where we can, in fact, realistically implement a farthest-in-future policy. That is, there can be an application where we receive a predetermined set of requests, so we know exactly what requests are coming down the pike.

But when we're leaving it up to human users to decide on the spot what their requests will be, the computer would have to be clairvoyant to take advantage of the farthest-in-future eviction policy. Therefore, in such cases, the clairvoyant eviction policy is only theoretical rather than something we can practically implement.

So, it seems that we're back to the drawing board. How can we design an eviction policy that is efficient even without knowing what requests will be made in the future?

LRU Cache

Although the computer cannot see into the future and know for certain what a user's next request will be, there are applications that enable the computer to *make an educated guess* as to what requests are upcoming. A number of cache eviction policies attempt to predict future requests based on looking at past requests. One such strategy is the *least-recently-used* eviction policy. In fact, least-recently-used, or LRU for short, is one of the most popular eviction policies in use today.

The term "LRU" describes which item we'll be evicting. That is, we will evict whichever item was *least recently used*; in other words, which item was least recently requested. For instance, if we have a cache containing the items `[1, 2, 3, 4]`, and our app keeps receiving additional requests for `1`, `2`, and `3`, but hasn't received a request for `4` in some time, it's the `4` that will be evicted from our cache next.

The idea behind LRU is that we're guesstimating that if we haven't received a request for item `4` in a while, it's likely because item `4` is becoming increasingly unpopular. And if item `4` is no longer popular, we're probably not going to receive a request for it in a long time, if at all.

Note that LRU is based on the farthest-in-future eviction policy in that our goal is to evict the item that will be requested farthest into the future. It's just that we use the idea of LRU to predict which item that'll be. In other words, if an item is least recently used, it's probably not in vogue right now and won't be requested again soon.

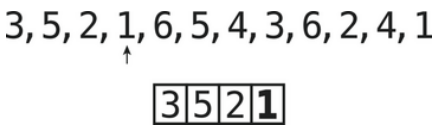
A cache that uses the LRU eviction policy is called an *LRU cache*. However, note that this name can be a bit misleading. It kind of sounds as if we'll be caching LRU items, while in fact the opposite is true; it's a cache *with an LRU eviction policy*. So LRU describes which items we *evict* from

the cache, not the ones we keep. And indeed, we *keep* the items that were *most* recently used.

LRU may not make sense in every application, but it's reasonable for many of them. This is because there are many contexts in which we can determine a request's popularity by how recently someone made that request.

LRU in Action

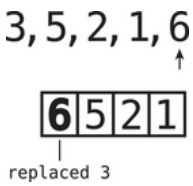
To see LRU in action, let's walk through an LRU cache example using the same request sequence we did when looking at the clairvoyant eviction policy. To speed things up, we'll jump to the step where we first filled the cache:



Okay, let's LRU!

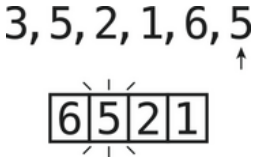
Our next request is a 6. This is a cache miss, so we must evict something from the cache to make space for the 6. With LRU, we look *leftward* to see which of the past items was least recently "used"—that is, least recently requested.

Of the items in the cache, the 3 was used least recently, so we evict it and replace it with the 6:

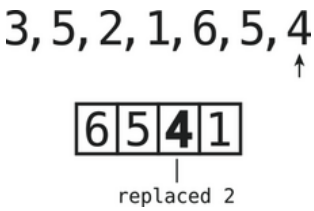


Note that I'm no longer displaying the complete list of requests to come, in line with the fact that we're not clairvoyant.

The next request is for the 5. We have a cache hit, which is gratifying:

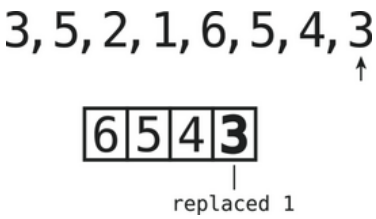


We now request a 4, which is a cache miss. Of the items in the cache, the 2 was least recently used, so we replace the 2 with the 4:

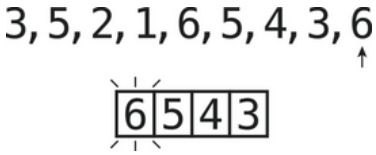


Next up, we have a 3. Ugh, we *had* the 3 in our cache a few steps ago, but we evicted it. Whatever, it's okay.

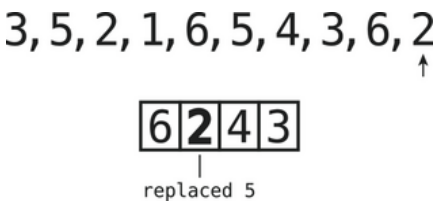
With this cache miss, we look at the items in the cache and determine which of them was least recently requested. Looking back at our previous request sequence, note that the 1 was used less recently than any of the other items in the cache. As such, we replace the 1 with the 3:



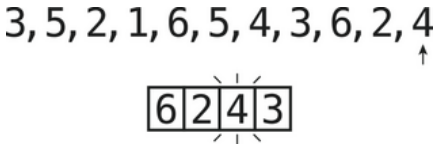
The next request is 6, and that's a cache hit. Sweet.



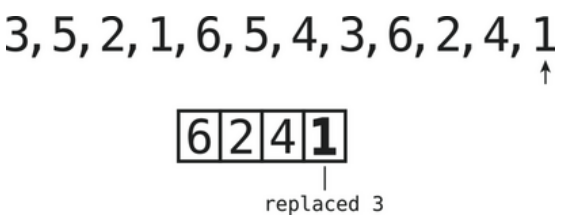
The 2 is up next. It's been some time since we've seen a request for a 2, so unfortunately, it's not in the cache. With this cache miss, we evict the 5, as it was least recently used:



Next up, we have a 4. Cache hit!



The last request is a 1, which is a cache miss. We evict the 3 since the 6, 2, and 4 were all more recently used:



If we count the cache misses here, including the initial requests used to fill the cache, we have a tally of 9. This isn't as good as the clairvoyant policy that yielded only 7 cache misses, but because the clairvoyant policy is only theoretical anyway, our more practical policy of LRU isn't half bad.

Now, the truth is that this example wasn't the ideal scenario for an LRU policy since LRU is essentially looking for trends, and no particular value was trending. The requests were kind of all over the place.

But the LRU policy *is* good for scenarios where requests trend and fade with time. In our cheapest-product search app example, we'd hope that if one person is searching for a particular product, this is a signal that the item is becoming popular, and other people will search for that same item.

Another practical example of this is a web browser. When one uses their browser to search the web, we might assume that the same user will revisit some of those same pages again in the near future. For this particular user, those pages are currently interesting and therefore "trendy."

So, that's the whole idea of LRU; we're taking a bet that current requests are trending, and because they're trending, we'll probably see those same requests made again soon.

Take the following sequence:

```
1, 1, 2, 1, 2, 1, 2, 3, 4, 3, 3, 4, 4, 3, 4, 3, 5, 5, 4, 4, 4,
4, 6, 4, 5, 6, 5, 5, 5, 6, 4, 5, 5, 6, 6, 5, 5, 7, 6, 5, 7, 7,
6, 7, 5, 6, 6, 7, 5, 5, 7, 7, 7, 8, 7, 8, 5, 8, 8, 7, 8, 8, 8
```

If you examine these numbers carefully, you might notice that items trend a little bit before the next "hot" number becomes popular. This is the ideal scenario that an LRU policy is designed for.

Care to take a guess at how many cache misses occur here if we use an LRU policy and our cache can hold up to 4 items? You could also take a pencil and paper and figure it out.

There are a whopping 63 requests here, but only 8 cache misses, and that includes the 4 cache misses that initially fill the cache. This sequence was practically *made* for LRU.

The LRU Cache Data Structure

This whole LRU thing sounds great in theory. But now we have to grapple with another issue: how do we track which items were least recently used?

In the previous walk-through of LRU, we looked back at past requests to see which ones were made recently and which ones hadn't been made in a while. Now, that would mean that we'd have to somehow save our past requests somewhere. But if we save all our requests, we defeat the entire purpose of an eviction policy, which is to free up memory! It doesn't help to evict items from the cache if we end up saving the same items somewhere else. A better approach is to somehow *indicate within the cache itself* which items were recently requested and which weren't.

There are several ways this can be done, but one clever tactic is to keep the cached items *sorted in the order of how recently used they were*. For example, if we have the cache `[1, 6, 4, 5]`, and we then have a cache hit with a request for the `4`, we take the `4` and move it to the front (that is, the left end) of the cache. The cache would then be sorted as `[4, 1, 6, 5]`.

And if the next request was for the `5`, which is also a cache hit, the cache would become `[5, 4, 1, 6]`. With this approach, the order of the items tells us how recently each item was used. That is, the item at the front of the list was used most recently, while the item at the back (that is, the right end) of the list is the LRU item.

Okay, so now we're getting somewhere. The question now is what data structure to use to implement this strategy. At the beginning of this chapter, we used a hash table to serve as our cache, and this was a sensible choice at the time. After all, with a hash table, we can read, write, and even evict in O(1) time. However, as discussed in Volume 1, Chapter 8, hash tables

cannot store values in sorted order. So, if we want to keep the cache values sorted by their recent use, a hash table comes up short.

An array, on the other hand, is *great* for keeping items sorted. But an array isn't a perfect solution either. This is because reading from the cache can take up to O(N) time because we'll have to perform a linear search on the array to find any value. That's way too slow for a cache; we ideally want to be able to read from the cache instantaneously. It turns out that no single classic data structure makes for a great LRU cache.

However, if we combine *two data structures together*, we can create an LRU cache whose time complexity is O(1) for all operations. Prepare to be amazed.

Data Structure Dynamic Duo

As I mentioned, a hash table is not capable of keeping values in any sorted order. However, we could consider using an array *in conjunction with* a hash table to keep track of the cache order. That is, we can store data in a hash table so we can access the data quickly, but we can *also* store a copy of that data in an array so we can track how recently it was used. Yes, we consume extra space by storing the same data twice over, but perhaps it's worth it. Let's see.

To analyze the efficiency of using an array and a hash table together, let's be absolutely clear on what operations we want our data structures to perform. At a high level, there are three major operations:

- *Reading from the cache.* Upon each and every request for data, we check whether the data already exists in the cache before bothering to fetch the data from an external source.

- *Managing a cache miss.* We have to evict data to make room for the new data.

- *Managing a cache hit.* We potentially have to update the order of the cache to indicate that the requested item is the most recently used.

Let's analyze how we'd perform these operations with a hash-table-and-array combo cache, starting with reading from the cache.

Read Efficiency

Again, we'll be performing our reads from the hash table since such reads happen in constant time. In fact, we don't need to touch the array at all when executing our reads. And so, here's the performance of our read operation:

	Read
Hash Table	O(1)

Cache Miss Efficiency

As we've seen, each time we read from the cache, we'll encounter either a cache miss or a cache hit. When we have a cache miss, we need to perform *two* cache operations. That is, we first evict the LRU item from the cache. Second, we insert the new data into the cache.

Now, here's the thing. Because we're modifying cache data, we need to update both of our data structures, namely, the array and the hash table. Specifically, we'll have to insert and delete data from each data structure. Fortunately, inserting and deleting from a hash table are each O(1) operations, so they'll run at breakneck speed. However, let's see how fast the array operations are.

First, we evict the *last* item of the array from the cache since we're sorting items so that the last element is the LRU item. Second, we insert the data we receive from the external source at the *front* of the cache. Being that the current request is, by definition, the most recent one, we put its data at the front to indicate that it is the most recently used item.

Here's a visual of these two cache operations:

Evicting the last item from an array takes O(1) time, but inserting at the front of the array takes O(N) time because we have to shift all the remaining values rightward. (This idea is covered in Volume 1, Chapter 1.)

Let's jot down the time complexity of all the cache operations we've looked at so far:

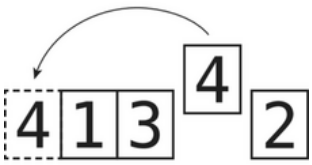
	Read	**Insert**	**Evict**
Hash Table	O(1)	O(1)	O(1)
Array		O(N)	O(1)

Incidentally, the fact that we decided that the most-recently-used item goes in the front of the array is arbitrary; we could have set things up in reverse. That is, we could alternatively put the most-recently-used item at the *end* while keeping the LRU item at the *front*. However, this doesn't help us in any way because although inserting at the end will now take O(1) time, evicting from the front will now take O(N) time. That is, when we delete an item from the front of an array, we then have to shift all the remaining values leftward.

Cache Hit Efficiency

The final operation we need to analyze is managing a cache hit. With a cache hit, we don't need to update the hash table in any way since we're not evicting anything, nor are we inserting any new data.

However, we do have to update the array. That is, we need to make sure that the data we just accessed gets moved to the front of the array to indicate that it's the most-recently-used data. This data might currently be anywhere in the array. It could be somewhere in the middle, or at the front, or at the end. If it's anywhere but the front, we need to pluck it from its current spot and move it to the front:



Here's how this breaks down in terms of time complexity. First, we have to *find* the requested data within the array before we can move it. This search can take up to O(N) steps. Second, we have to move this item to the front of the array, causing the other elements to shift positions. If, for example, we move the last element to the front, this causes N-1 shifts, making this move another O(N) operation.

In our analysis going forward, we'll keep these "Find" and "Move to Front" operations separate as shown in the following table:

	Find	**Move to Front**
Array	O(N)	O(N)

Each operation has the potential to take O(N) time.

Complete Efficiency Analysis

Okay, we've completed our efficiency analysis. Here's the complete table of the speed of our proposed hash-table-and-array-combo cache:

	Read	**Insert**	**Evict**	**Find**	**Move to Front**
Hash Table	O(1)	O(1)	O(1)		
Array		O(N)	O(1)	O(N)	O(N)

It's hard to say whether this is "good" or "bad" since we don't have any other solutions currently on the table. However, if we want to improve upon this approach, we need to find ways to reduce the time of one or more of these operations.

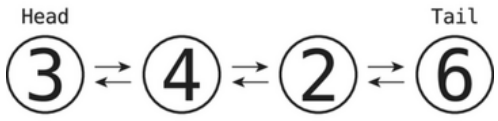
Currently, the room for improvement lies within most of the array-based operations. As mentioned, one issue is that we're inserting and deleting from both ends of the array, but an array can only act fast on one end; the other end takes O(N) time. Is there another data structure that maintains order but can also quickly insert and delete data on *either* end?

Linked Lists to the Rescue

As discussed in Volume 1, Chapter 14, a classic linked list allows us to insert and delete data from the list's head in O(1) time. While this seems promising, recall that linked lists insert and delete data from the tail in O(N) time. Ultimately, a classic linked list won't serve us better than an array since we're looking for a way to insert and delete quickly from *both* ends of the cache.

However, a *doubly* linked list *can* insert and delete data from both ends in constant time. Again, this is because we track both the head and tail of the list at all times and can thereby access both ends instantaneously.

Here's a simple depiction of a doubly linked list serving as a cache:



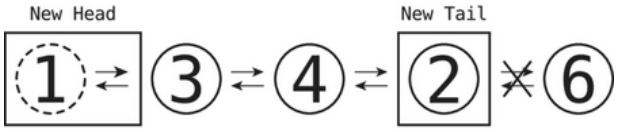
Again, we'll continue to read from the hash table rather than the doubly linked list since reading from a hash table is just O(1). Reading from the doubly linked list would take up to O(N) since we'd have to perform a linear search to find anything.

Let's now take a look at the doubly linked list's operations and their efficiency.

Linked List Cache Miss

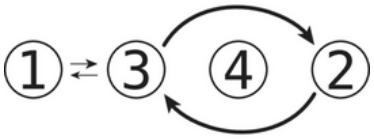
When we encounter a cache miss, we can evict the tail in a single step because doubly linked lists always track the tail. We can now also insert a new item at the head of the list in one step because linked lists also track the head.

In the following diagram, we insert item 1 at the head and also evict item 6. To evict the 6, we set the 2's "next" link to `None` (rather than the 6) and declare the 2 to be the list's tail going forward.

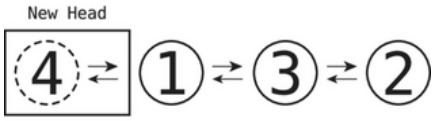


Linked List Cache Hit

When we chance upon a cache hit, we still have to spend N steps to find the item we're looking for, but when we do, moving it to the front can be done in O(1) time. Specifically, we remove the desired node by having its two adjacent nodes link to each other, which effectively disconnects the desired node from the list:



We then take that node and make it the head of the list:



Linked List Advantages

It turns out that using a linked list instead of an array significantly boosts the speed of our cache! (From here on, I'll sometimes refer to the doubly linked list as simply the "linked list.")

Here's a summary of the time complexity of our current linked-list solution:

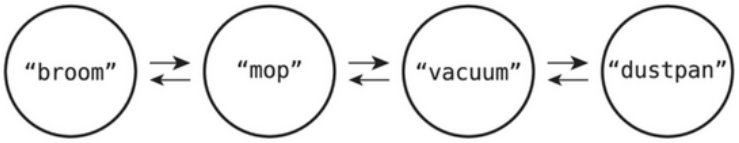
	Read	**Insert**	**Evict**	**Find**	**Move to Front**
Hash Table	O(1)	O(1)	O(1)		
Doubly Linked List		O(1)	O(1)	O(N)	O(1)

All of our cache operations occur in constant time except for finding cache data in the linked list when we have a cache hit, which takes up to N steps. However, with a clever trick, we can even get the "Find" operation down to constant time.

Point to the Node

Following is a high-level visual of what our cache system currently looks like, using products and their prices as sample data:

```
{"broom": 99, "mop": 129, "vacuum": 299, "dustpan": 79}
```

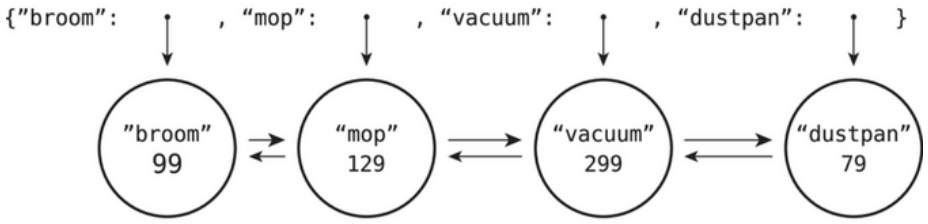


As you can see, we store the data in duplicate: one set in the hash table, and the other inside the linked list. Note that we don't need to store the prices themselves in the linked list; we just need the list to tell us the recency of when each product was requested.

In the diagram, we listed the hash keys in the same order as the linked list to make the visualization easier to grasp. However, keep in mind that hash

tables have no inherent order; we're relying solely on the linked list to keep the cache data sorted.

Currently, finding data in the linked list during a cache hit takes O(N) time. But with one small but arguably mind-blowing adjustment, we can get this "Find" operation to run in constant time. And that is, instead of storing the raw data inside the hash table, *we store the nodes of the linked list* in the hash table instead. In other words, the keys of the hash table remain the same, but the values will no longer be integers; the values will be the very nodes of the linked list:



What an interesting data structure! Although the nodes are contained as values within the hash table, there's no reason why the nodes cannot still link to each other and thereby form a linked list. Note that for convenience, we're now also placing the price data inside the nodes themselves.

As with our previous cache implementations, we still begin every request by reading the hash table to check whether the requested data is currently in the cache. But now, each time we have a cache hit, we don't have to search the linked list for that data. Instead, the hash table itself points the way *directly* to the appropriate node of the linked list since the node is the actual value in the hash table. And once we have that node in hand, we'll move it to the head of the linked list.

With this modification, our LRU cache achieves O(1) speed for *all* of its operations:

	Read	**Insert**	**Evict**	**Find**	**Move to Front**
Hash Table	O(1)	O(1)	O(1)		

	Read	Insert	Evict	Find	Move to Front
Doubly Linked List		O(1)	O(1)	O(1)	O(1)

I'll highlight more of the nitty-gritty details of each of these operations in the code walk-through that follows.

Code Implementation: LRU Cache

To code up our LRU cache, we'll need a doubly linked list, so let's take a look at a doubly linked list implementation. This implementation differs slightly from the one demonstrated in Volume 1, Chapter 14 since now we're focusing on the functionality needed to serve our cache.

We'll start by looking at the code for the nodes themselves:

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next_node = None
        self.previous_node = None

        if isinstance(data, dict):
            self.product = data.get("key")
            self.price = data.get("data")
```

This node is a double-ended node, having links to both the `next_node` and `previous_node`. Additionally, I've added some custom `product` and `price` attributes for the sake of our product price-searching app.

I've saved this code in a file called `double_ended_node.py`, and now import it into the implementation of the doubly linked list. Here's the doubly linked list code:

```
import double_ended_node
```

```
class DoublyLinkedList:

    def __init__(self, first_node=None, last_node=None):
        self.first_node = first_node
        self.last_node = last_node

    def append(self, data):
        new_node = double_ended_node.Node(data)

        if not self.first_node:
            self.first_node = new_node
            self.last_node = new_node
        else:
            new_node.previous_node = self.last_node
            self.last_node.next_node = new_node
            self.last_node = new_node

        return new_node

    def insert_head(self, data):
        new_node = double_ended_node.Node(data)

        if not self.first_node:
            self.first_node = new_node
            self.last_node = new_node
        else:
            new_node.next_node = self.first_node
            self.first_node.previous_node = new_node
            self.first_node = new_node

        return new_node

    def pop_head(self):
        popped_node = self.first_node
        self.first_node = self.first_node.next_node
        self.first_node.previous_node = None
        return popped_node

    def pop_tail(self):
        popped_node = self.last_node
        self.last_node = self.last_node.previous_node
        self.last_node.next_node = None
        return popped_node
```

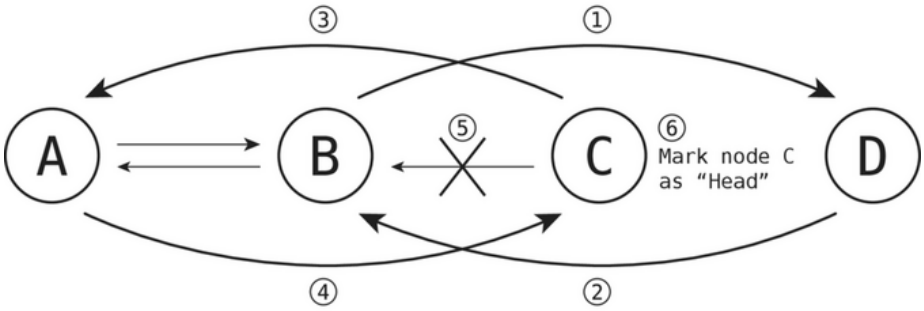
```
    def move_to_head(self, node):
        if node == self.first_node:
            return

        if node.next_node:
            node.previous_node.next_node = node.next_node
            node.next_node.previous_node = node.previous_node
        else:  # node is the tail
            node.previous_node.next_node = None
            self.last_node = node.previous_node

        node.next_node = self.first_node
        node.next_node.previous_node = node
        node.previous_node = None
        self.first_node = node
```

The `append` and `pop_head` methods were already present in the Volume 1 implementation, but we've added a few new methods to ensure that the linked list can function as part of an LRU cache.

The `insert_head` method is similar to the `append` method, except that `insert_head` inserts a new node at the beginning of the list, whereas `append` inserts at the end of the list. Similarly, the `pop_tail` method is similar to `pop_head`, except that `pop_tail` removes and returns the list's tail rather than its head.

The `move_to_head` method, which moves a given node to the head of the list, contains a number of steps and may feel a bit like performing a surgical operation. The following image highlights the various links we need to add and remove if the node we're moving is currently somewhere in the middle of the list.

Say that we have a list in which the nodes are A, B, C, and D. If we want to move C to the front of the list, we make the following moves:



1. Change B's `next_node` to point to D.
2. Change D's `previous_node` to point to B.
3. Change C's `next_node` to point to A.
4. Change A's `previous_node` to point to C. (Previously, A was the head and so its `previous_node` pointed to `None`.)
5. Set C's `previous_node` to `None` since C will be the head.
6. Mark C as the official `head` of the linked list.

At the end of the day, our list will appear like this:

With our linked list in place, we can now introduce our `LruCache` implementation:

```
import doubly_linked_list
import time


class LruCache:

    def __init__(self):
        self.hash_table = {}
```

```
        self.linked_list = doubly_linked_list.DoublyLinkedList()
        self.max_size = 4

    def read(self, key):
        if key in self.hash_table:  # Cache hit
            return self.freshen(key)
        else:                       # Cache miss
            return None

    def freshen(self, key):
        node = self.hash_table.get(key)
        self.linked_list.move_to_head(node)
        return node

    def cache(self, key, data):
        # If cache is full:
        if len(self.hash_table) >= self.max_size:
            self.evict()

        # Save new data in both linked list and hash table:
        new_node = self.linked_list.insert_head({"key": key, "data": data})
        self.hash_table[key] = new_node

    def evict(self):
        # LRU eviction policy:
        evicted_node = self.linked_list.pop_tail()
        del self.hash_table[evicted_node.data["key"]]

class PriceRequester:

    def __init__(self):
        self.cache = LruCache()

    def request_price_for(self, product):
        data = self.cache.read(product)
        if data:  # Cache hit
            price = data.price
        else:     # Cache miss
            price = self.search_web_for(product)
            self.cache.cache(product, price)

        return price
```

```
    def search_web_for(self, product):
        # Mock data:
        price = 1
        # Mimic time it takes to search web:
        time.sleep(0.25)

        return price
```

Two classes are at play here. The more important class is the `LruCache`, which serves as a generic LRU cache. But we've also included a `PriceRequester` class as an example of an application that uses the cache. While I recommend that you glance at the `PriceRequester` code, for now, we're going to only walk through the `LruCache` class itself. Let's take it from the top.

We've saved our doubly linked list code in a file called `doubly_linked_list.py`, so our code imports that module. We also import the `time` module to mock a web request within the `PriceRequester` code as we did earlier in the chapter.

Let's take a look at the constructor of the `LruCache`:

```
def __init__(self):
    self.hash_table = {}
    self.linked_list = doubly_linked_list.DoublyLinkedList()
    self.max_size = 4
```

Here, we create the cache's hash table and doubly linked list. We also set the maximum number of cache items to be `4`, but this can easily be changed to any other number.

The `read` method attempts to find an item from the cache by looking up the item in the hash table:

```
def read(self, key):
    if key in self.hash_table:
        return self.freshen(key)
    else:
        return None
```

Reading from the cache yields either a cache hit or a cache miss.

Upon a cache hit, we call the `freshen` method, whose details I'll walk through shortly. The primary purpose of the `freshen` method is to move the item's corresponding linked-list node to the front of the list to indicate that this item is the most recently used item.

If we have a cache miss, though, we return `None` to indicate that the item is not presently in the cache. This means that our app will have to find the data from an external source like the web, after which it can cache that data.

Next up, we have the `freshen` method, which moves a given node to the head of the linked list and ends by returning that node:

```
def freshen(self, key):
    node = self.hash_table.get(key)
    self.linked_list.move_to_head(node)
    return node
```

The next method in this class is the `cache` method, which stores data inside the cache:

```
def cache(self, key, data):
    if len(self.hash_table) >= self.max_size:
        self.evict()

    new_node = self.linked_list.insert_head({"key": key, "data": data})
    self.hash_table[key] = new_node
```

Before caching any new data, the `cache` method first checks to see if the cache is already full. This is determined based on the `max_size` attribute defined in the class's constructor. If the cache *is* full, we evict an item from the cache. (I'll cover the `evict` method shortly.)

To cache new data, we first create a new node and place it at the head of the linked list to indicate that it's the most recently used item. Then, we add the

item's key to the hash table and set the value to be the node we created for the linked list.

The final method, `evict`, removes data from the cache according to the LRU eviction policy:

```
    def evict(self):
        evicted_node = self.linked_list.pop_tail()
        del self.hash_table[evicted_node.data["key"]]
```

We need to evict the data from both the linked list and the hash table. Since the LRU node is the tail of the list, that's the node we delete.

To remove the corresponding key from the hash table, we take a look at what item our deleted node contains and look for that item's key in our hash table. We then delete that key-value pair from the hash table.

Fixing the LRU Worst-Case Scenario with Randomization

We've encountered many algorithms that perform differently based on best, average, and worst-case scenarios. Sometimes, the worst-case scenarios are pretty bad. But other times, worst-case scenarios can be *really* bad. The worst-case scenario for an LRU cache is, in fact, a nightmare.

This worst-case scenario can occur when performing a nested loop in which the inner loop iterates over a list of items where the number of items is *ever so slightly* greater than the number of items that the cache can store. Let me explain what I mean.

Continuing with our product price-searching app example, say that on the app's homepage, we want to display some great deals from the web. Also, say that we want to show one deal at a time in a carousel widget, cycling through a list of deals.

One way to do this is to have an array of product names and have an infinite loop that iterates over the array again and again. For each product, the app will look up that product's best price, pause for a few seconds, and then move on to the next product from the list.

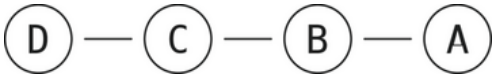
Here's a simplified implementation of what I'm talking about:

```
import time


products = ["broom", "mop", "vacuum", "dustpan", "sponge"]

while True:
    for product in products:
        price = search_web_for(product)
        print("Great Deal!!! Get a " + product + " for just " + str(price))
        time.sleep(5)
```

Because we're cycling through the same list of products over and over again, we don't need or want to perform a web search each time we display a deal. Instead, it would be smarter to look up the price once and cache it. This way, we can pull the price from the cache the next time we display that product again.

There's nothing wrong with this approach *unless* we encounter the nightmare scenario. This is when the product list's length is slightly greater than the size of the cache. Let's see what happens when we continuously iterate over five items, and our cache can contain only a maximum of *four* items. To make the diagrams simple, we're going to call the products A, B, C, D, and E.

After we fetch data for the first four items in the order A, B, C, and D, our cache is full:



If our next step fetches E, which is not in the cache, we insert E and evict the LRU item A:

We've completed our first cycle through the products, so our loop restarts and fetches the same items again. This means that next up, we're going to fetch the A. However, we just evicted the A in the previous step, which is pretty unfortunate. As such, we'll have to fetch the A from the web again. Oh, and when we do, we'll also have to evict B, which is the LRU item:

Our next request is the B.

Wait, what? It's not in the cache? We had the B in the cache a second ago! Okay, whatever.

Next in line is to fetch the C. But … we evicted the C from the cache the moment before we needed it.

This pattern, of course, continues forever, rendering the cache utterly useless. We consistently evict each item from the cache the step before we need it! And so, we never—and I mean *never*—get to pull data from the cache. It's not official jargon, but I call this the "LRU trap."

Of course, our product-searching example is somewhat contrived, but the LRU trap can and does happen to unwitting software developers every day. The question is how we can have efficient caching while also avoiding the LRU trap, which is a cache's worst nightmare.

LRU + Randomization = Sweet Dreams

Fortunately, there is a solution. If we throw some randomization into the mix, we may be able to achieve the balance of fast caching while also avoiding the LRU trap.

I mentioned earlier that LRU is one of many eviction policies that have been described over the years. Some alternative eviction policies, for example, use randomization as part of the eviction algorithm. One such policy is known as *random replacement* and is completely different than LRU. Instead of tracking the cache data in any way, we evict data at random.

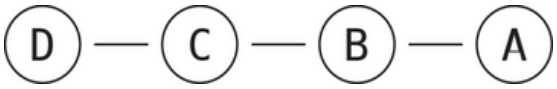
This approach absolutely avoids the LRU trap since we're not evicting data according to any pattern, let alone the LRU trap pattern. However, random replacement turns out to be a pretty inefficient eviction policy. The whole point of a clever eviction policy is to predict what data might be requested next, and random replacement doesn't bother to predict anything whatsoever.

Power of Two Choices Strikes Again

However, we may be able to *blend* randomization *with* LRU to create an eviction policy that is predictive and also avoids the LRU trap at the same time.

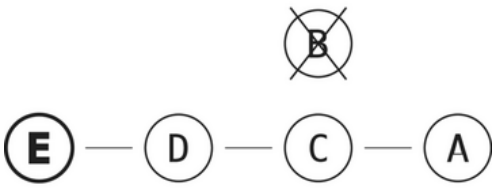
One such blend utilizes the same *power of two choices* discussed earlier in *Randomized Load Balancing with the Power of Two Choices*. That is, we randomly select *two* items from the cache and evict whichever item is less recently used. So, we're not necessarily evicting the cache's least-recently used item, but we're evicting the least-recently used item *from among two random choices*.

Again, let's refer back to our sample cache:



If our next request is for item E, we randomly choose two cache items. Let's say our "dice" land on C and B.

Between C and B, B is less recently used, so that's the one we evict, as shown in the figure.



Although we're using some randomization, we still leverage some of LRU's ability to predict the future. That is, between the choices of C and B, C is more likely to be requested again, so we keep it in the cache.

Next up, we request the A. This time, the A *is* inside the cache. Cache hit! Phew, we avoided the LRU trap.

Indeed, this eviction policy may not be as good at predicting future requests as full-fledged LRU, but its predictions may be good enough and also avoid the LRU trap.

Evicting a Random Node

The power-of-two-choices approach has another drawback: evicting a random node from a linked list is slower than evicting the tail node, which is what we were doing with the classic LRU cache.

Again, because our linked list is a *doubly* linked list, we are able to evict the tail in O(1) time since a doubly linked list always has immediate access to the tail in addition to the head. Evicting a *random* node, though, can take up to O(N) time. Although there's more than one way to delete a random node, they all require traversal of the list.

In any case, let's implement this power-of-two-choices LRU cache.

Code Implementation: Power-of-Two-Choices LRU Cache

Here's the strategy we'll use to evict a node: we'll randomly pick two numbers from 0 up to the length of the list and use these numbers to represent the indexes of nodes. We'll then select whichever of the two numbers is greater since this will represent the node closer to the tail and therefore less recently used. We'll then traverse the list up to the index we've selected and remove that node.

To make this all work, we'll add the following `pop_index` method to the `DoublyLinkedList` class:

```
    def pop_index(self, index):
        if index == 0:
            return self.pop_head()

        current_node = self.first_node

        # Traverse the list while counting up to the desired index:
```

```
        for _ in range(index):
            current_node = current_node.next_node

        # If the index corresponds to the tail, simply pop the tail:
        if current_node == self.last_node:
            return self.pop_tail()

        # If the index corresponds to a node that is not the tail, delete
        # the node by updating the links of the node's neighbors:
        current_node.previous_node.next_node = current_node.next_node
        current_node.next_node.previous_node = current_node.previous_node

        return current_node
```

This method accepts a given index and pops the corresponding node. For example, if the index is 0, we pop the first node. If the index is 2, we pop the third node.

We then modify the `evict` method from our cache as follows:

```
    def evict(self):
        # Randomized LRU eviction policy:
        random_1 = random.randint(0, self.max_size - 1)
        random_2 = random.randint(0, self.max_size - 1)
        node_index_to_evict = max(random_1, random_2)

        evicted_node = self.linked_list.pop_index(node_index_to_evict)
        del self.hash_table[evicted_node.data["key"]]
```

This selects the greater of two random indexes and pops the node at the greater index, just as we said we'd do.

The Memory Hierarchy

Throughout this chapter, I've discussed caches that are used to store data that is pulled from an external source such as the Internet. However, caches are used in many other contexts as well, including in the hardware of our computers. In fact, understanding how this hardware works can enable us to write more efficient day-to-day code. So, let's talk hardware.

It's generally understood that a computer has *memory*. A computer's memory is where a computer stores all its data, from entire files to the tiny variables we create with our code. However, our computer doesn't simply store all data in one huge container. Instead, a computer's memory is partitioned into a number of different *levels*. And here's the interesting thing: each level has a *different speed* at which it offers up data. Some levels of memory provide data at blazing speeds, while other levels are relatively slower at letting the computer access data.

In a digital utopia, *all* levels of memory would be as fast as possible. One major reason why this is not the case is that the fastest memory costs a lot of money. That is, the fastest memory technology available at any given moment in time is relatively expensive. If the computer's entire memory was made up of that technology, the computer's cost would be prohibitive. So, in the real world, computers are built so that they have some levels of memory that are, indeed, extremely fast, but other levels that are slower.

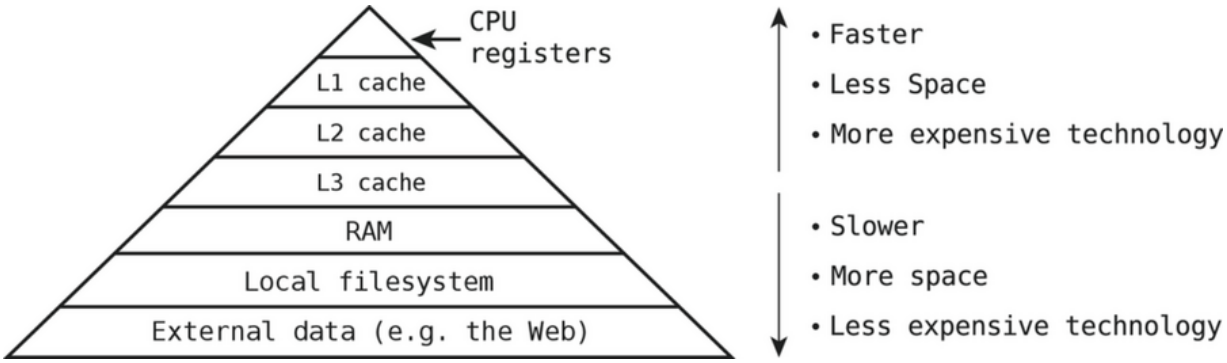
Now, the trick here is this: any piece of software only needs a limited amount of data at any given time. For example, most software doesn't usually interact with every single file on your computer and certainly not all at the same time.

Along these lines, think about the code you write day-to-day. Often, your code works with a mere handful of variables at a given point in time. Now, we want our software to run quickly, so the computer stores these variables

in the fastest levels of memory. While these fast levels of memory are made of expensive technology, the costs are kept down by making these levels small. That is, they only store a relatively small amount of data. But that's fine since that's all the data your software currently needs access to.

All the other files on our computer—which our software *doesn't* need to access at the moment—can live in cheaper, slower memory levels. Because these slow levels are cheaper, we can afford to make them large and capable of storing large amounts of data.

This system, in which our computer contains a combination of small-but-fast memory levels in addition to large-but-slow memory levels, is known as the *memory hierarchy*. The memory hierarchy is often visualized as a pyramid, like this:



The top levels of the pyramid contain only a small amount of data, but are made of expensive technology that offers up data at blazing speeds. As you move down the pyramid, each level has increased storage capacity but is also somewhat slower since the slower levels are made of cheaper hardware.

Let's briefly walk through each level, starting from the bottom.

The Memory Hierarchy Levels

The bottommost level isn't truly a level of memory *inside* the computer. It refers to data that is external to your computer, such as the web. However,

it's often included in the description of the memory hierarchy, so I threw it in there as well.

The Filesystem

The next level up is the filesystem on your computer, which is where all those files like photos, music, and book reports live. Those files can all live in this slow memory level until you access them with an app like a photo viewer, music player, or word processor. For the record, the filesystem is also sometimes called the *hard drive*, or the *hard disk*, or simply the *disk*.

Main Memory

Here's where things get interesting. When you do open, say, your photo-viewer app, this app will *load a photo from the filesystem into the next level up* of the memory hierarchy. This level is sometimes called *main memory* or *random access memory* or simply *RAM* for short. I'll use each of these terms interchangeably.

The key thing to understand here is this: generally, most code doesn't interact directly with files in the file system. Instead, the software first loads data from the filesystem into main memory and *then* interacts with the data in main memory. Indeed, if the data is being modified, at some point these changes will be written back to the filesystem. But in the short term, a computer processes data that is within main memory (or the higher levels of the memory hierarchy, which I'll get to soon).

Now, when you close your app, all this data gets cleared from RAM. Hopefully, you saved your changes! If you did, this data will live on in the filesystem until you need to access it again. It's important to be cognizant of the size of your RAM. Even if you have a large *hard drive*, the amount of data your code can process at once can be no larger than your *main memory*.

If you see an ad for a computer with, say, 32GB *memory* and 1TB *storage*, this means that the *main memory* can store up to 32 gigabytes of data, and

that the *filesystem* can store up to 1 terabyte of data. A terabyte is 1,000 gigabytes, which is way larger than the 32GB of main memory, but that terabyte is only going to store your filesystem. Your code, on the other hand, won't be able to process more than 32GB at once. And if you have other software running at the same time, some of that 32GB is already being utilized by those other applications, so your code has even less than 32GB to work with.

In short, don't create a variable that tries to hold a massive amount of data unless you know that the computer has enough RAM to handle it. Later, in Chapter 7, I'll discuss how to handle situations where you *do* have to work with massive data like that.

Cache Levels

Software doesn't only interact with the main memory level; the computer has a few levels that are even faster than RAM. These levels, which are the top four levels of the memory hierarchy pyramid, act as caches, albeit in a slightly different way than how I've described caches until now. Let me explain.

Say that your code loads an array of one million integers from a file (in the file system) and stores the array in a variable. Once this happens, this array will live somewhere inside main memory.

Now, suppose that your code iterates over the array to perform some computation, such as getting the total sum of the integers. At this point, your computer may take, for example, the first 1000 integers and copy them from RAM to the L3 cache. The reason it does this is that the L3 cache is made of faster technology than RAM. So, when the computer iterates over these integers, it can do so more quickly.

The only reason the computer doesn't load the *entire* array into the L3 cache is that the L3 cache is *too small* to hold them all. So, what the computer does is load the first 1000 integers into the L3 cache and then,

after iterating over them, it evicts them and loads the *next* 1000 integers from RAM into L3. The computer then iterates over them and then evicts them to load the next 1000 integers. The computer repeats this process until it's iterated over all the array's integers.

Note that the numbers I'm using, such as one million and 1000, are examples; every computer model stores different amounts of data in their various cache levels.

Previously, I described a web cache as being a location for holding data from an external and slower source, such as the Internet, so that we can access the data more quickly in the future. In the current context, the L3 cache is holding data from an *internal* but slower source—in this case, RAM—so that it can process the data more quickly. The goal of both caches, though, is the same: to move data from a slower location to a faster location. Like all caches, your computer's cache levels all have eviction policies. The policy may be LRU or something else; it all depends on the computer model.

Much of the time, the computer does all this caching without you being aware of it. The computer figures out what data your code is processing right now and moves it into the appropriate cache levels.

Not every type of computer is exactly the same, but most computers have multiple cache levels, such as L1, L2, and L3. Each of these caches serves as a cache for the level below it. So, for example, the computer might copy 1000 integers into L3, and of those, 100 integers into L2, and of those, 10 integers into L1. Again, these numbers are all just examples.

CPU Registers

The CPU registers act as a cache as well. In fact, it's the fastest of all caches because it's made of the most expensive technology. Accordingly, it's also the smallest of all the caches. You can think of the CPU registers as being the "L0" cache.

It turns out that understanding these concepts can allow you to write faster code. Let’s see how.

Writing Cache-Friendly Code

Let's get to the fun stuff. While it seems that we've merely had a leisurely tour through some of our computer's hardware, these concepts can directly impact the way we should write code.

Following is some code that computes the sum of integers contained within a two-dimensional array. For simplicity, let's assume that the array is square, meaning that the rows and columns have equal lengths. I'm going to present two versions. Here's Version One:

```
def compute_sum(array):
    size = len(array)
    sum = 0

    for row_index in range(size):
        for column_index in range(size):
            sum += array[row_index][column_index]

    return sum
```

It's pretty straightforward. We iterate over each row, and within each row, we iterate over each column, adding up the numbers as we go.

Now, here's Version Two, which is only subtly different:

```
def compute_sum(array):
    size = len(array)
    sum = 0

    for column_index in range(size):
        for row_index in range(size):
            sum += array[row_index][column_index]

    return sum
```

It's almost identical to Version One except that now we're going in column order first, meaning that our outer loop iterates over the column indexes and

the inner loop iterates over the row indexes.

Both versions get the job done, but let's visualize the difference between these two versions.

In Version One, the following diagram depicts the order we'd process, say, an array of size 3. In the diagram, I placed little numbers in circles to indicate the order in which we iterate over each integer. (I also made all of the array integers `9` so we don't get distracted with all the different numbers floating around.)

```
[
  ① ② ③
  [9, 9, 9],
  ④ ⑤ ⑥
  [9, 9, 9],
  ⑦ ⑧ ⑨
  [9, 9, 9],
            ]
```

And here's the order of Version Two:

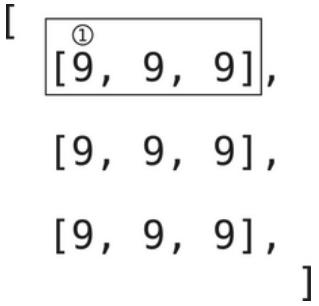
```
[
  ① ④ ⑦
  [9, 9, 9],
  ② ⑤ ⑧
  [9, 9, 9],
  ③ ⑥ ⑨
  [9, 9, 9],
            ]
```

Believe it or not, there's a significant efficiency difference between these two versions since iterating by row first is faster than iterating by column first. The reason for this is the caching concepts we've just discussed.

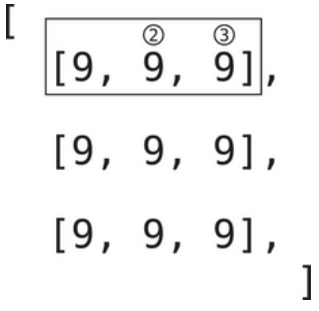
We saw that if code sets or accesses a variable, that variable will be cached inside the CPU register or the like. But here's something important: if you access an item from an array such as `array[0]`, the computer doesn't only cache `array[0]`. Depending on the array's size, the computer may cache the *entire* array, or at least a good chunk of it.

This makes sense because we often iterate over arrays. And if we start a loop by accessing `array[0]`, it's clear that we're about to also access `array[1]` and `array[2]` soon. So the computer, smartly, caches a large chunk of the array so that we can access those later indexes super quickly.

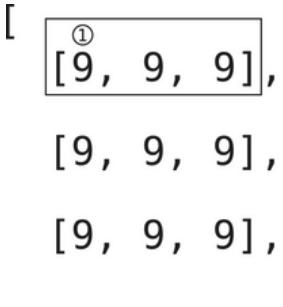
So in Version One, our inner loop first accesses `array[row_index][column_index]`, which initially is `array[0][0]`. Because the computer accessed the first inner array, the computer "smartly" caches that entire array. In the following diagram, I used rectangles to indicate what is currently cached:



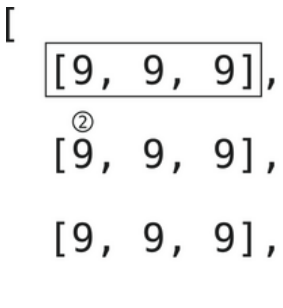
Now, this is great since next we'll be accessing `array[0][1]` and `array[0][2]`. Because the entire `array[0]` is cached, the computer can grab all of these values almost instantaneously:



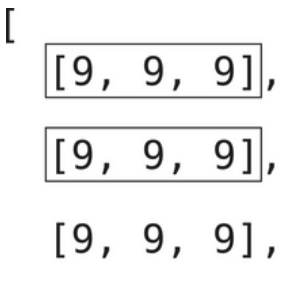
However, in Version Two, where we go in column order first, here's what happens. In this version, we also start at `array[0][0]`, so the entire `array[0]` is cached like before:



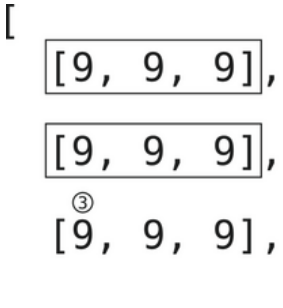
But here's the snag. We next access `array[1][0]`, which is *not* inside the cache:



As such, the computer doesn't benefit from the cache speed when reading `array[1][0]`. Once we do access `array[1][0]`, though, the computer now caches the entire `array[1]`:



Next, the computer looks up `array[2][0]`. Man, this isn't in the cache either!



As "smart" as the computer is, it can't *always* predict your next move. Its array-caching system is ideal if we follow the rows first, but backfires when

we go by columns first.

Spatial Locality

Computer scientists like to refer to this idea as either *spatial locality* or *locality of reference*. That is, code is faster if it keeps accessing data that is near other data that was accessed recently. So, if our code accessed `array[0]`, it's great if the next data the computer grabs is `array[1]` or `array[2]`. Because `array[1]` and `array[2]` are near `array[0]`, they're likely already in the cache.

Understanding the computer's caching hardware allows us to avoid these pitfalls and write faster code. In this case, the difference is significant. When I benchmark summing a two-dimensional array of size 10,000, Version One takes about 9 seconds, while Version Two takes roughly *18* seconds. Leveraging spatial locality, in this case, makes our software twice as fast!

Arrays vs. Linked Lists

Another practical application of memory caching comes into play when iterating over arrays and linked lists. Although iterating over each data structure takes O(N) time, it's way faster to iterate over an array than a linked list. This, again, is because of spatial locality. Specifically, when we access `array[0]`, the next chunk of the array is loaded into the cache for quick access.

With a linked list, though, the nodes are not necessarily near each other in memory. Therefore, when we access the list's first node, the other nodes do not get cached. The computer has to laboriously jump around from memory cell to memory cell to access each node. Accordingly, iterating over a linked list is generally slower than iterating over an array.

Mergesort vs. Quicksort

Ah, you thought we were done talking about Mergesort, weren't you? Surprise! Back in *Comparing Mergesort and Quicksort: Lessons Learned*, I explained that even though both Mergesort and Quicksort, in the world of Big O notation, are O(N log N), Quicksort is usually faster than Mergesort in the real world.

Back then, we offered up one potential reason for why this might be so. But now we have another possible reason: Quicksort is better than Mergesort in terms of spatial locality.

As different computers utilize caches in different ways, it's not always easy to definitively explain why one algorithm works better with a cache than another. However, here's a general idea: because Quicksort sorts an array in place, the algorithm repeatedly accesses the same array. So, once this array is in the cache, Quicksort can always access each element of the array extremely quickly. Mergesort, on the other hand, is always creating *new* arrays. Because Mergesort has to juggle data from a number of different arrays, not all the data that we need at the moment is in the cache.

Wrapping Up

Caching is truly a pivotal concept in computing. Whether you'll be implementing your own caches or taking advantage of the memory caches within the memory hierarchy, knowing how caches work enables you to push the envelope of your code's speed.

In the chapters that follow, we're going to continue to explore the theme of randomization and specifically the concept of *randomized data structures*. Randomized data structures are data structures whose whole power stems from random numbers. The first such data structure we'll discuss is the *treap*. But before we dive into the world of treaps, in the next chapter we'll first contrast treaps with their nonrandomized counterparts—red-black trees, a classic must-know data structure.

Exercises

The following exercises provide you with the opportunity to practice with caching. The solutions to these exercises are found in the section *Chapter 4*.

1. Say that we have a cache that holds up to 5 values. How many cache misses occur for this sequence if our cache uses the clairvoyant eviction policy?

   ```
   "c", "t", "h", "o", "p", "t", "h", "z", "o", "a", "p", "t", "b", "z",
   "h"
   ```

2. How many cache misses will occur if our cache uses the *LRU* eviction policy for the prior sequence?

3. Following is a Python class representing a rather contrived concept that I call a "Bit Box":

   ```
   class BitBox:
       def __init__(self):
           self.red_bits = [1] * 10000
           self.blue_bits = [1] * 10000
           self.green_bits = [1] * 10000
   ```

   Basically, each Bit Box is a storage container for the integers `0` and `1`. Each Bit Box contains an array of 10,000 "red" bits, 10,000 "blue" bits, and 10,000 "green" bits. When each Bit Box is created, all the bits are set to `1`.

   The next bit of code creates a single Bit Box:

   ```
   bit_box = BitBox()
   ```

   Following are two different methods that count all the `1` bits in a Bit Box. Which of these methods has better spatial locality?

   ```
   def count_bits_1(bit_box):
   ```

```
    sum = 0

    for i in range(10000000):
        sum += bit_box.red_bits[i]
        sum += bit_box.blue_bits[i]
        sum += bit_box.green_bits[i]

    return sum


def count_bits_2(bit_box):
    sum = 0

    for i in range(10000000):
        sum += bit_box.red_bits[i]
    for i in range(10000000):
        sum += bit_box.blue_bits[i]
    for i in range(10000000):
        sum += bit_box.green_bits[i]

    return sum
```

4. *Puzzle:* Here's an exercise where it might help to combine two data structures together to produce an efficient solution. (Yes, that was a hint.)

   Create a data structure that allows for O(1) searches, O(1) insertions, but *also* allows for O(1) random samples. In this context, a random sample means that we pick at random a single value from the data set. As always, each value must have an equal chance of being chosen. The trick here is how to achieve O(1) for reads, insertions, *and* random samples.

Chapter 5

The Great Balancing Act of Red-Black Trees

One of the most fundamental and useful data structures out there is the binary search tree, or BST for short. (I covered BSTs at length in Volume 1, Chapter 15.) However, as I pointed out in *Binary Search Tree Randomization*, BSTs lose their special powers if they become imbalanced.

In this chapter, I'll introduce you to *self-balancing BSTs*. These are BSTs that use clever algorithms to ensure that the tree never becomes too imbalanced. The main focus of this chapter is the *red-black tree*, a widely used self-balancing BST that is also one of the classic data structures of computer science. You'll discover how red-black trees work, *why* they work, and how to create your own basic red-black tree from scratch.

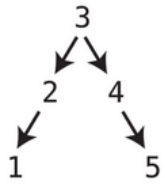
Red-black trees have a certain notoriety for being complicated, and while this chapter is indeed on the long side, I'll lay everything out for you in a way that makes it all easier to digest. So, let's jump right in.

Online Algorithms and Self-Balancing Trees

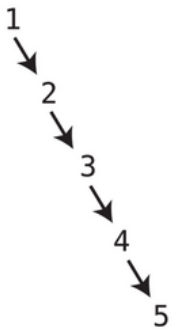
In Volume 1, Chapter 15, I noted that the primary benefit of BSTs is that they maintain data in order while *also* providing fast searching, insertion, and deletion. While hash tables are faster than BSTs when it comes to searching, insertion, and deletion, hash tables do not maintain any order. So, if you want ordered data that is also decently fast when it comes to the other operations, BSTs are a fantastic solution.

However, as I mentioned, BSTs have a worst-case scenario to contend with. When we insert values into a BST *in order*, the BST becomes imbalanced. Here's a quick refresher.

If we build a tree by inserting data in the order of `3, 2, 4, 1, 5`, we get this BST:



Beautiful. It's as balanced as my dad's checkbook. However, if we insert the data in perfect order, namely, `1, 2, 3, 4, 5`, we get:



This BST is *imbalanced* and is as slow as a linked list for searching, insertion, and deletion. In fact, it pretty much *is* a linked list.

Earlier, in Chapter 3, *How Random Is That?*, we developed a simple solution for this problem—we randomized the data! That is, before building the BST, we shuffled all the data, making it highly unlikely that the data will be inserted in order. But here's the thing: this is a great solution for *some* applications, but it doesn't satisfy *all* applications.

The randomized solution only works if we have all the data upfront before we begin building our BST. That is, say that we have an array of integers ready to be converted into a BST. Indeed, all we have to do is shuffle the data beforehand and, the BST is likely to be balanced when we build it.

But now say that we *don't* have all the data in front of us. For example, perhaps we're building and maintaining our BST over a long period of time. Suppose that our app receives its first integer at 1:00 and creates the initial BST. Then, at 2:00, we receive another integer and add it to the BST. At 3:00, we receive another integer and insert it into our tree.

In such a scenario, we don't have the ability to randomize the data before we insert it into the tree because we don't have all the data before creating the tree. The tree has already been built, and if we now receive a new integer, there's no way to shuffle that integer together with the data already in the tree. In such a case, if our application receives ordered data, our BST will be imbalanced.

Online Algorithms

This is an example of an *online algorithm*, which is an algorithm that deals with data that is received in bits and pieces over an extended (and possibly infinite) period of time. The term "online" here can be a bit misleading, as it sounds like it has something to do with the Internet. It does not. In this context, "online" simply means that the data is still arriving, and we have to run an algorithm now, even though we don't have all the data yet. In other words, an online algorithm is an algorithm that has to process data that is continually arriving.

You saw an example of an online algorithm in the previous chapter. Caches and their associated algorithms often process data that's received over a period of time.

Conversely, an *offline algorithm* is one that only begins working once it has all the data in hand. Most of the algorithms we've dealt with before fall under this category.

So, with regard to BSTs, if they need to be "online" and process data continually as it comes in, randomization is not a solution. However, there's another solution. It's called the self-balancing tree.

Self-Balancing Trees

A *self-balancing tree* is one that, well, balances itself! As data is inserted or deleted, the tree rearranges its nodes so that the tree remains balanced or at least reasonably so.

Here's a straightforward example of a self-balancing tree that you should never, ever use. I'm making it up off the top of my head to make a point. I call it The Phoenix Tree.

The Phoenix Tree is a BST that executes the following steps any time we insert or delete data:

1. The tree copies all of its own data into an array.
2. It shuffles the array.
3. It (figuratively) burns the original tree down to the ground and rebuilds the entire tree from scratch using the randomly ordered data.

Out of the ashes of the old tree emerges a new one. Because we've randomized the data before building the tree anew, there are good odds that the tree will be decently balanced. So, it's fairly easy to make a self-balancing tree. The tricky part, though, is to make a self-balancing tree that is *efficient*.

The Phoenix Tree is—with no offense to phoenixes—an efficiency disaster. After all, we're constantly rebuilding the entire tree from scratch. To put it in Big O terms: inserting into The Phoenix Tree takes at least O(N) time since we process all N elements each time we insert a single item. Compare this with a regular BST, where insertion is a speedy O(log N).

What we need to do is develop a self-balancing tree that doesn't just work, but is also efficient.

Red-Black Trees

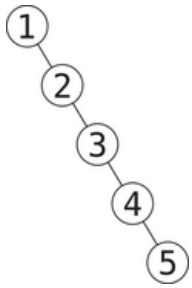
The need for an efficient self-balancing tree is precisely why computer scientists developed the *red-black tree*. The tree bears this name because its nodes are colored either red or black, for reasons we'll look at shortly. Using brilliant algorithms, red-black trees keep themselves balanced and do so without taking too much extra time.

While these trees are clever and fast, they're complex and involve many details. Most presentations of red-black trees plow straight into these details, getting into the weeds right away. This usually leaves people terribly confused.

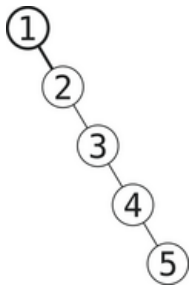
As such, I'd like to start not with the *what*, but rather with the *why* and the *how*. Like peeling an onion, we'll unpack this topic one layer at a time. This should make red-black trees considerably easier to grasp.

Rotations—Part 1

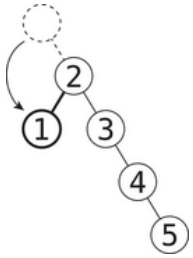
Let's start with a visual example of how red-black trees maintain balance. Look at this terribly imbalanced tree:



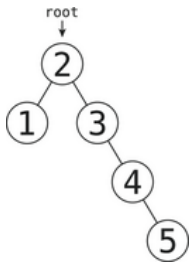
In the following diagram, I highlight one particular segment of the tree:



Now, watch what happens when I *bend* that segment ever so delicately:

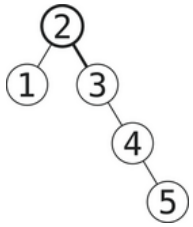


The 2 becomes the root instead of the 1:

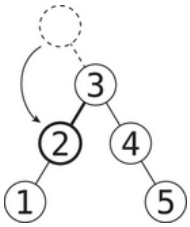


The tree is now a little more balanced than before.

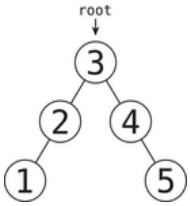
That was fun! Let’s do it again. Here, I highlight another tree segment:



And watch what happens when I bend that segment:



The 3 becomes the new root, and voila! The tree is now perfectly balanced:



This bending technique is known as performing a *rotation*. That is, we rotate segments of the tree. In this example, I happened to rotate two different tree segments. And if you look carefully, you'll see that I rotated them in a counterclockwise direction.

This is a general idea of how rotations work (as rotations have a whole bunch of additional details to them). The gist is that as we insert and delete values from the tree, we perform these rotations when necessary to maintain the tree's balance.

However, it's not so easy to tell the computer when and how it should perform these rotations. This is especially true when dealing with a large, complex tree. We need a set of algorithms that list the cases for when a rotation is necessary and provide instructions on how to perform the rotation in each case. On top of this, we need to do this quickly. Accordingly, the computer shouldn't have to analyze the entire tree to calculate how the rotation is to be executed.

To make this task less daunting, clever computer scientists found a way to make it so that these algorithms can work in bite-sized chunks. That is, upon each insertion and deletion, only a small subset of the tree needs to be analyzed, and a minimum number of rotations need to be performed. To accomplish this, they proposed a self-balancing tree that contains *two types*

of nodes: one red and one black. Over time, you'll see how this setup helps our situation, but let's first look at the nature of this tree.

A Tale of Two Nodes

As mentioned, red-black trees get their name from the fact that they have two types of nodes that we treat as being either "red" or "black." Let me explain what that means.

Here's some basic code representing a node:

```
class Node:
    def __init__(self, value):
        self.value = value
```

We can turn this node into either a "red" node or a "black" node by adding a new attribute to it:

```
class Node:
    def __init__(self, value, color):
        self.value = value
        self.color = color
```

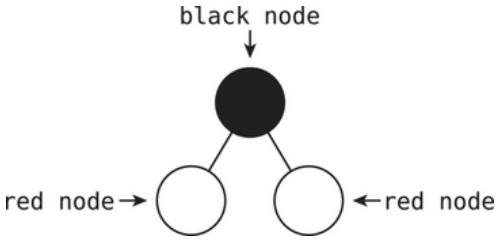
We can now assign to the node's `self.color` a string value of either `"red"` or `"black"`.

The idea behind assigning a color to a node is simply so that we have a mechanism for designating two different types of nodes. (We'll see soon *why* this is important.)

Now, to create two different types of nodes, we could have used *any* type of attribute. For instance, we could have assigned nodes different shapes, or different sizes, or different letters. We could even assign nodes different characters from *The Lord of the Rings* if we were feeling so inclined. Whatever attribute we use, the main goal is to have two different types of nodes. True, nodes are already different in that they contain different `value`s, but we're looking to also put all nodes into two different general categories.

And so, we have color. The colors didn't have to be red and black; they could have been anything. I'm pretty sure, though, that they chose red and black instead of *white* and black to annoy authors who publish books with images that are only black and white.

And that brings me to my next point. In this book, the diagrams represent red nodes using *white*-colored nodes like this:



Here's a pro tip: if you wear red-tinted glasses while reading this book, the images will look *perfect*.

The thing to address, now, is what we gain by having different types of nodes. When a tree has different types of nodes, we can construct algorithms that only have to analyze and deal with small sections of the tree rather than having to analyze the entire tree, which naturally would take time. Instead, the algorithm can analyze the section of the tree where it inserted or deleted a node, and by examining the pattern of surrounding nodes and their colors, it can perform appropriate rotations on that section of the tree.

As you'll see later in this chapter, a tree segment may have various patterns, such as a black node with two red child nodes, or vice versa. The algorithm will know what to do with each pattern, including whether a rotation is in order, and if so, how to execute the rotation. As you may imagine, there are a number of possible patterns, and that's exactly what makes a red-black tree complex; we need our algorithm to handle all of these patterns. And, sometimes, performing a rotation may form a new pattern that will require yet *another* rotation.

But truly, these concepts aren't difficult in themselves. It's just that there are a lot of details. Luckily for you, I spell them all out right here.

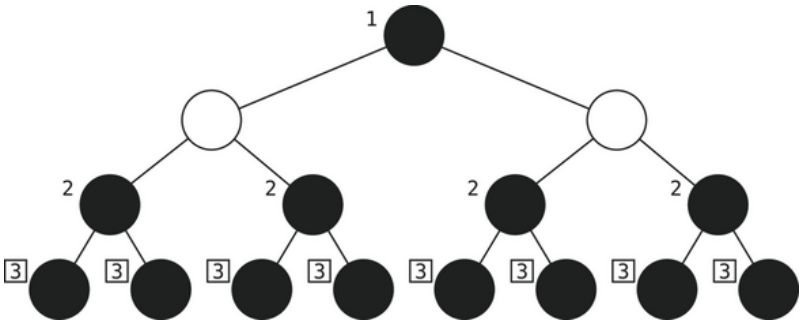
The Red-Black Rules

As mentioned, a red-black tree consists of nodes that are either red or black. There's no way around this; each node must choose a side. You can't have a node that has no color at all, or a node that is both red *and* black, or a node that prefers to be mustard yellow. Each node is either red or black, period.

However, not every tree that consists of red and black nodes is deemed a valid red-black tree. There are two major rules, which I call "The Red-Black Rules," that a tree must adhere to for it to be admitted into the official red-black tree club:

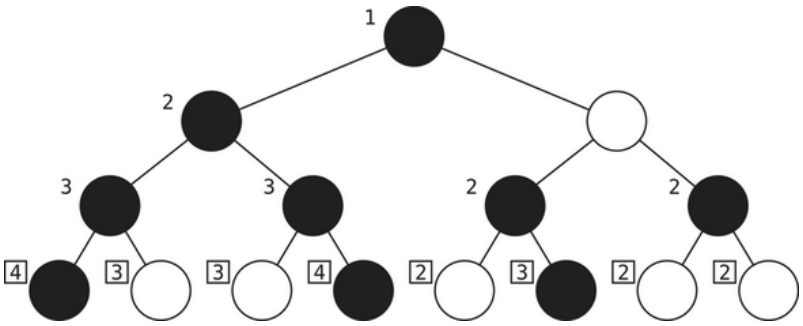
- *The Black Height Rule:* Each path from the root node to the bottom of the tree must contain the same number of black nodes.
- *The Red Enemies Rule:* A parent and child cannot both be red. Think of red nodes as being angry enemies; you can't put them next to each other!

The Black Height Rule takes its name from a concept known as *black height*. Take a look at the following red-black tree:



If you follow the path from the root node down to each *leaf* (a node that has no children, and is the final node of a path), you'll see that each path has three black nodes. The number by each node in this illustration shows how many black nodes there are from the root to that node. This means that each path in our tree has a *black height* of 3.

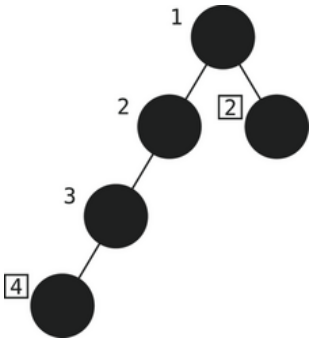
The following tree is an *invalid* red-black tree since different paths have varying numbers of black nodes:



In this tree, some paths contain two black nodes, while others contain three black nodes, and yet others contain four black nodes. This violates the Black Height Rule, and the tree is, therefore, invalid. It may be red and black, and it may be a tree, but it's not a red-black tree.

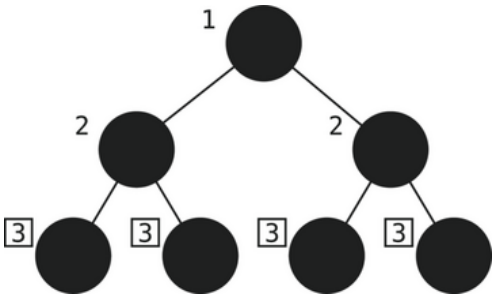
The Black Height Rule is the driving force behind ensuring the tree's balance, and here's why.

Imagine for a moment that there was no such thing as red nodes, and instead, our tree could only contain black nodes. The Black Height Rule enforces that a tree can never be imbalanced. Take this imbalanced tree:



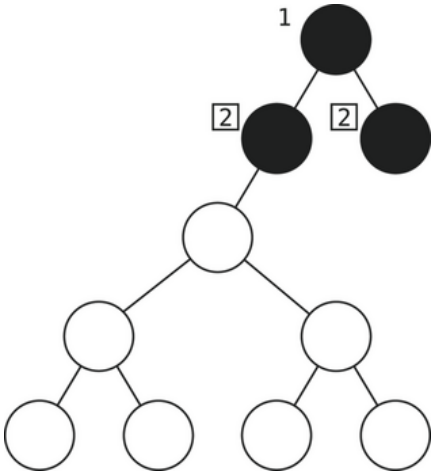
This tree is imbalanced *because* its two paths have different black heights. The Black Height Rule would never allow this to happen. By definition, a tree with only black nodes where all its paths have the same black height is going to be balanced.

However, if we continue in this imaginary world where our tree can only contain black nodes, we quickly run into a major problem. Take a look at this perfectly balanced tree:



If we want to insert a new black node, where can we put it? We can't put it anywhere since the insertion will cause one path to have a greater black height than the other paths! And this is why red-black trees also contain red nodes. Inserting a red node into a tree will never directly cause a violation the Black Height Rule, and it will allow us to grow the tree.

On the other hand, it would seem that the existence of red nodes can pull the rug from under the feet of the Black Height Rule. That is, while the Black Height Rule is trying to ensure that each path of the tree has the same black height, we can still make a tree highly imbalanced, like this:

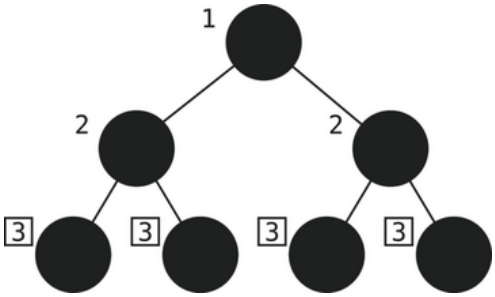


This, my friends, is why the Red Enemies Rule exists. Again, the Red Enemies Rule states that we can't have a parent and its child both be red, so this tree is invalid.

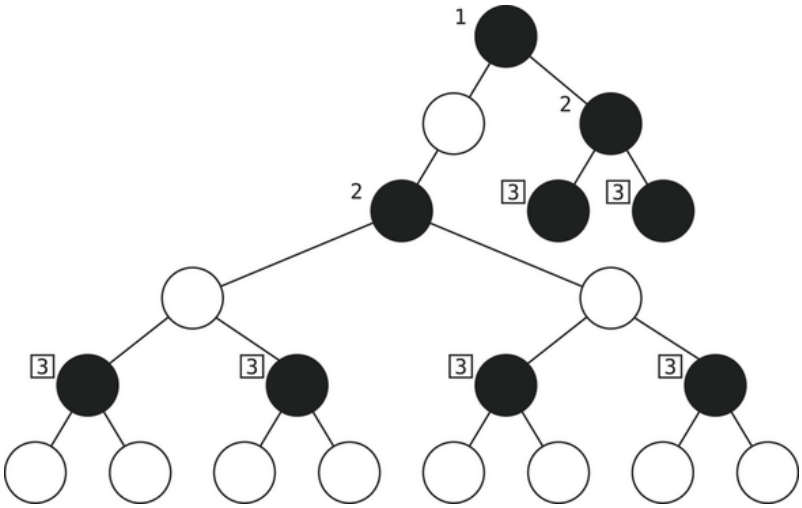
So, the Black Height Rule and Red Enemies Rule work *in tandem* to create a balanced but flexible tree. The Black Height Rule enforces balance, while the existence of red nodes provides some flexibility for tree growth. At the same time, the Red Enemies Rule prevents an abundance of red nodes from making the tree imbalanced.

Balanced Enough

A red-black tree does not always maintain *perfect* balance, however. Let's see how imbalanced we can make a red-black tree, even while adhering to our two rules. Here, again, is a perfectly balanced tree:



Let's try to add red nodes to make this tree as imbalanced as possible. What I came up with is shown in the tree.



Here, I extended the left branch of the tree by inserting red nodes between all the black nodes. And I didn't violate any rules! Each path has the same black height, and we don't have any red nodes connected to each other.

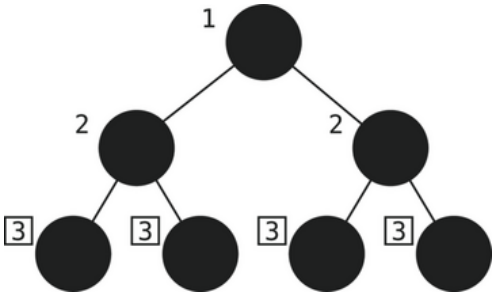
Indeed, a red-black tree is not designed to be *perfectly* balanced. Rather, it is designed to be balanced *enough*. While "balanced enough" sounds like some sort of subjective term, we can give it a firm and objective definition. That is, the goal of a red-black tree is to guarantee that it's impossible to make one path *more than twice as long as another path*.

Here's why this is so. If, with any red-black tree, I were to try to make one path longer than the others, I can't add more black nodes without violating the Black Height Rule. And I also can't string a bunch of red nodes together without creating red enemies. Instead, my only option is to *insert a red node between every other black node*. At most, then, I can double the length of one path over the others, but I can't push it any further.

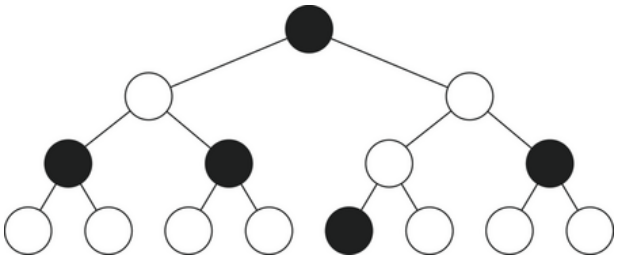
Ultimately, the combined push and pull of the Red-Black Rules make it so that a red-black tree remains pretty well-balanced and yet flexible enough to insert and delete nodes.

Phantom Nodes

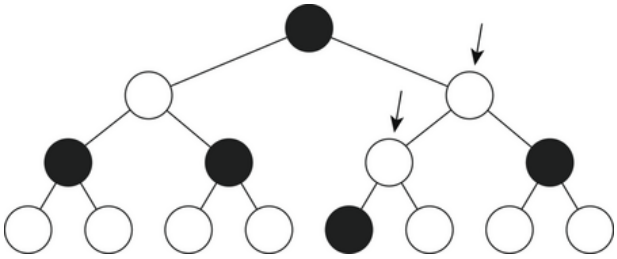
Okay, pop quiz time! Is the following tree a valid red-black tree?



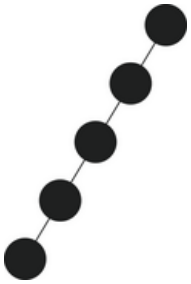
It seems hard to call this a red-black tree when it doesn't contain any red nodes. Yet, it does not violate any of our rules. All paths have the same black height, and there certainly aren't any two adjacent red nodes. So it is, indeed, valid. How about this one?



If you answered that it's invalid, you're correct since it has a pair of red enemies:



And now, I'm going to throw a monkey wrench into everything with a trick question. (Sorry!) Is the next tree a valid red-black tree?



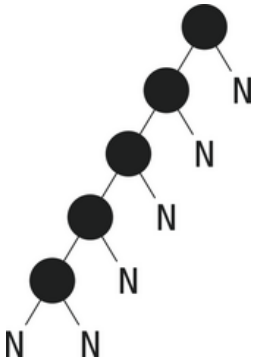
It certainly has no adjacent red enemies, that's for sure. And we don't seem to have different paths with different black heights because apparently there is only one path. So the tree *seems* valid. However, this is pretty disturbing. We learned that the Red-Black Rules try to ensure balance, but this tree is the ultimate imbalanced tree! What has the world come to?

The answer to this is based on a concept known as *phantom* nodes. (Spooky!) When analyzing a red-black tree, we have to fill the tree in with imaginary, *phantom* nodes. Here's what I mean.

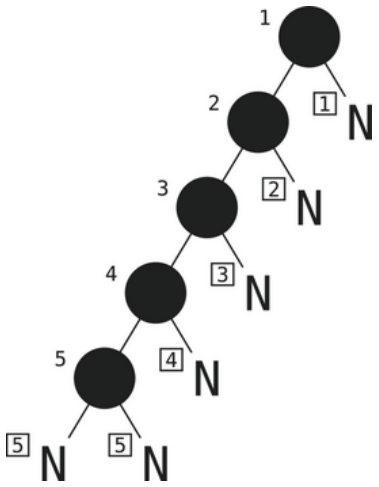
Each node in a BST has the ability to hold two children: a left child and a right child. However, in a typical BST, some nodes have two children, some

have one, and the leaves have none. In a red-black tree, in each spot where a node could *potentially* have a child but does not, we fill that spot in with an imaginary node. Let me demonstrate with an example.

In the tree from the previous diagram, the top four nodes each only have a left child, and the leaf has no children at all. This means that there are a number of empty spots in the tree where a child could exist but does not. Specifically, the top four nodes have empty spots for a right child, and the bottom node has two empty spots for both of its potential children. Therefore, we populate all the missing spots with phantom nodes. In the following image, a phantom node is represented with the letter N, representing `None`:



In your code, you don't need to implement phantom nodes (although some people do). However, the reason why these phantom nodes are important is that they totally change the way we look at black height. That is, without phantom nodes, this tree has only one path and doesn't violate the Black Height Rule. But *with* the phantom nodes, the tree now has multiple paths since *each path from the root to a phantom node is considered a viable path*. With this in mind, here are the black heights of the tree's paths:



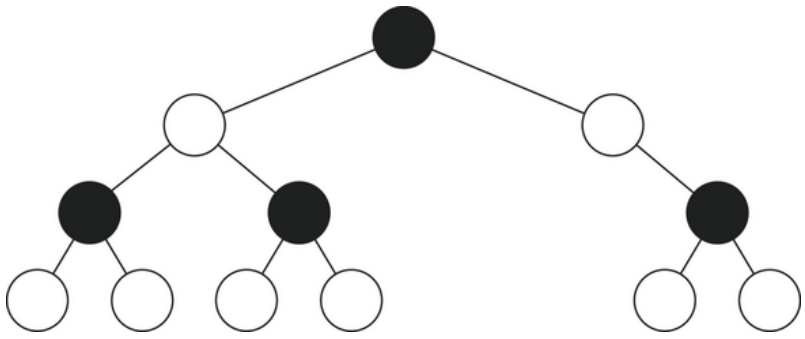
Note that the phantom nodes themselves do not increase a path's black height. However, each phantom node represents a path's destination.

Now we can see that this tree violates the Black Height Rule many times over. There are six paths, and almost all of them have different black heights. We can now breathe a sigh of relief since the Black Height Rule indeed will prevent such an imbalanced red-black tree from ever existing.

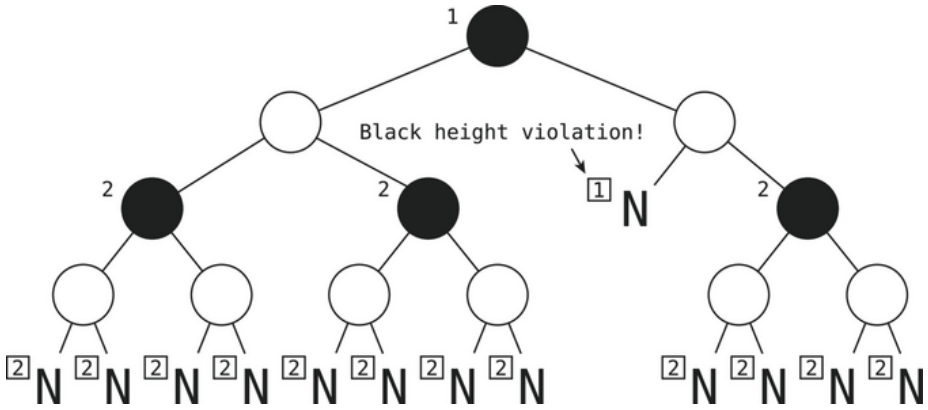
Technically, then, the Black Height Rule should be restated to accommodate phantom nodes. Here goes:

- *The Black Height Rule:* Each path from the root node to any *phantom node* must contain the same number of black nodes.

Let's look at one more example to solidify all of this. Is the following tree a valid red-black tree?



It is not. When the phantom nodes are revealed, we'll see that while most paths from the root to each phantom node have a black height of 2, there's one path that has a black height of 1:



I omit these phantom nodes in most of the diagrams that follow. However, they're important to keep in mind when calculating the black height of a tree's path. Again, some people do implement phantom nodes in their code, but it's not necessary. Our implementation—which we'll get to eventually—will leave them out so as to keep the code more concise. We can get away with this because the red-black tree insertion and deletion algorithms don't bother to count a path's black height. Instead, the algorithms follow a different set of patterns, as we'll soon see.

On that note, some of the literature states that there's a convention to always make sure that the root of a red-black tree remains black. That is, it should start out as black, and if it somehow becomes red (you'll see later in this chapter how this can happen), we should color it black again. However, this is only a convention, and not necessary. Once again, we'll ignore it to simplify our code. But it's good for you to know that it exists.

Rotations—Part 2

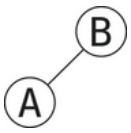
So far, I've mentioned two general ideas regarding how a red-black tree maintains its balance. One is the technique of rotation, in which we bend the tree in advantageous spots. The other is the Red-Black Rules. These two concepts work together, and here's how.

Each time we insert or delete a node from the tree, we often cause a violation of the Red-Black Rules. (You'll see examples of this shortly.) To get the tree back in compliance with the rules, we rotate sections of the tree.

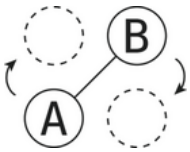
In other words, a tree that follows the Red-Black Rules will always be reasonably balanced. But to get our tree to comply with these rules, we often have to perform rotations. And that's how the Red-Black Rules and rotations relate to each other.

I described rotations in a general way in *Rotations—Part 1*. Here, though, I delve into precisely how rotations work.

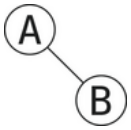
Whenever we perform a rotation, we focus on two nodes: a parent and a child. These nodes can be anywhere in the tree, but the diagrams that follow zoom in on these two nodes. Take the following example, with a parent node called B and a left child node called A:



Let's rotate this tree segment clockwise:



This leaves us with:

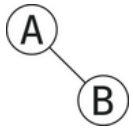


As you can see, a rotation does two major things:

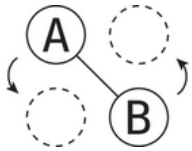
1. What was the parent now becomes the child, and what was the child now becomes the parent.

2. The left-right orientation of the parent-child relationship switches. In this example, the child was originally a *left* child, but what is currently the child is now a *right* child of its parent.

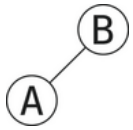
Rotations can also go counterclockwise. As you're about to see, they're a mirror opposite of a clockwise rotation. Here's an example in which B is the *right* child of A:



Whenever we rotate a parent and child, the determination as to whether it should be clockwise or counterclockwise depends on whether the child is the left or right child. When it's the left child, as in the previous example, we perform a clockwise rotation. When we have a *right* child, as in the present scenario, we perform a *counterclockwise* rotation:

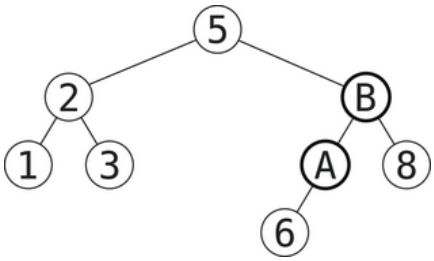


This gives us:



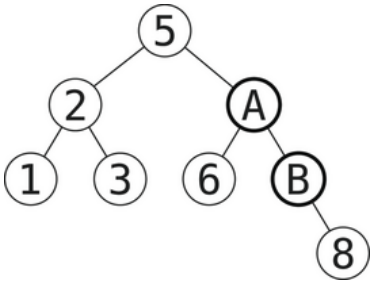
Note that after we rotate the nodes counterclockwise, they end up in the state that the nodes looked like before a *clockwise* rotation. Similarly, after we rotate the nodes clockwise, they end up in the state that the nodes looked like before a *counterclockwise* rotation. In theory, we could take the same two nodes and rotate them back and forth forever, but of course, that would be completely pointless.

Let's see what this all looks like in the context of a larger tree such as this one:



Although most of the nodes hold integer values, I snuck A and B nodes into the tree as well. Let's focus on those two nodes since those are the ones we're going to rotate.

Because A is B's *left* child, we execute a clockwise rotation:



A is now B's parent, and B has become the right child of A. Note that the 6, which was A's left child, is still A's left child even after the rotation. Similarly, the 8 has never changed from being B's right child.

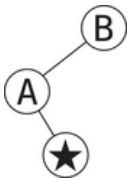
Also, note how we've "pulled" the 6 closer to the top of the tree, while "pushing" the 8 down. In this case, the tree isn't any more balanced than before. However, if the 8 didn't exist, this rotation would have transformed the tree from being a four-level tree into a three-level tree, and also made the tree perfectly balanced.

Keep in mind that since a red-black tree is a type of BST, not only does the tree have to adhere to the Red-Black Rules, but it also has to follow the rules of any BST. The main rule regarding a BST is that for each node of the tree, the node's left descendants must all have smaller values than it, and the node's right descendants must all have greater values than it.

One of the key things to know about rotations is that they never interfere with this BST rule. No matter how many segments you rotate clockwise or counterclockwise, the tree will always remain a valid BST. As such, we can perform rotations to fix Red-Black Rule violations without ever having to worry that we may inadvertently introduce a new BST rule violation.

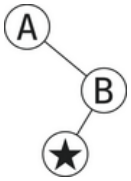
Crossover Nodes

There's one more detail about rotations that's important to know. To help illustrate the concept, I'll pose a conundrum: how would you rotate the A and B nodes in the following tree? Grab a pencil and paper and try to do this yourself before moving on.



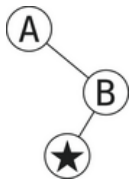
You've learned that we need to rotate these nodes in a clockwise fashion because the parent, B, has a left child. But here's the thing: with a clockwise rotation, B will become A's right child. However, A already *has* a right child—the star! So, the problem is what we're supposed to do with the star node.

In such cases, rotations perform one more switcheroo: specifically, we designate the star to be a *crossover node*. That is, the star crosses over by completely detaching itself from the A, and becoming a left child of B:

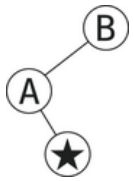


Now, there's no reason to worry about what happens if B already has a left child because A *was* B's left child! So, by definition, when we make it so that A is no longer B's child, that automatically opens up a spot for B to

have a new left child. The same applies to a counterclockwise rotation, but in reverse. Take the following tree, for example:



We run into a problem when trying to turn the A into the B's left child since B already has the star left child. So, we pull off the same crossover node trick as before by making the star become A's right child:



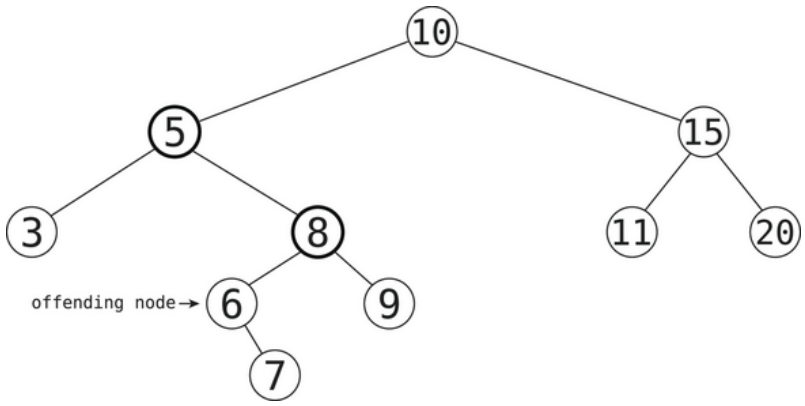
To sum it all up, a rotation involves either two or three changes to the tree:

1. The child and parent switch places.

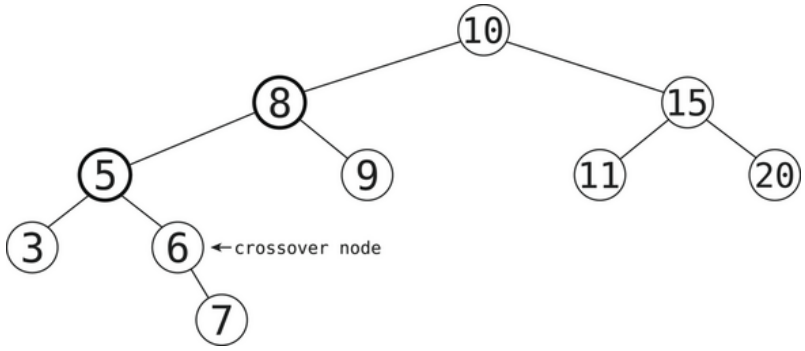
2. We switch the orientation of the parent-child relationship. If we had a left child, we now have a right child, and vice versa.

3. If the new parent (say, A) already has a child (say, the star), where the new child (say, B) is supposed to go, we turn the offending node (the star) into a crossover node and make it a child of B, with the opposite orientation. That is, if the crossover node was a right child of A, it now becomes a left child of B, and vice versa.

Let's put this all together in the context of a realistic tree where there are no letters or stars, but only integers. In the following diagram, we're going to rotate the 5 and the 8:



The 8 is the 5's right child, so we're going to rotate counterclockwise. However, the 6 is an offending node because it's blocking us from turning the 5 into the 8's left child. So we have the 6 cross over and become the 5's right child:



Note that the crossover node's children come along with it. In this example, the 7 remains the right child of the crossover node just as before.

Okay, we're almost ready to tackle inserting new values into a red-black tree. But because rotations play a crucial role in these algorithms, let's implement the rotations first.

Code Implementation: Nodes and Rotations

Here's an implementation of a red-black tree node:

```
class Node:
    def __init__(self, value, color):
        self.value = value
        self.color = color
        self.left_child = None
```

```
        self.right_child = None
        self.parent = None
```

As with a regular BST node, we've established attributes for the node's `value`, `left_child`, and `right_child`. And because we're dealing with a red-black tree, we've also added the critical `color` attribute that we've previously described.

However, you'll also note that this code includes a `parent` attribute as well. While this wasn't necessary for the regular operations of the classic BST, it makes the implementation of a red-black tree's operations more straightforward, as you'll see soon.

The `parent` attribute allows each child node to point to its parent. This is akin to a doubly linked list, where each node links not just to its next node, but also to its previous node. Here as well, each node links both to its children and to its parent. Each node will have a `parent` except for the root node, whose `parent` will remain `None`.

Let's begin implementing the actual `RedBlackTree`. This class will contain a lot of code by the time we're done, but here's the basic class plus its rotation operations:

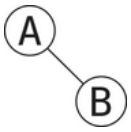
```
import rbt_node


class RedBlackTree:
    def __init__(self, root=None):
        self.root = root

    def rotate_counterclockwise(self, a, b):
        # "a" is the parent, and "b" is the right child
        a.right_child = b.left_child

        if b.left_child:
            a.right_child.parent = a
```

```
        b.parent = a.parent
        if not b.parent:
            self.root = b
        elif b.parent.left_child == a:
            b.parent.left_child = b
        else:  # "a" was a right child
            b.parent.right_child = b

        b.left_child = a
        a.parent = b

    def rotate_clockwise(self, b, a):
        # "b" is the parent, and "a" is the left child
        b.left_child = a.right_child

        if a.right_child:
            b.left_child.parent = b

        a.parent = b.parent
        if not a.parent:
            self.root = a
        elif a.parent.right_child == b:
            a.parent.right_child = a
        else:  # "b" was a left child
            a.parent.left_child = a

        a.right_child = b
        b.parent = a
```

The class constructor initializes the tree's `root`, which represents the root node of the tree. Like a classic BST, we'll need to keep track of this at all times.

We then have the rotation methods, `rotate_counterclockwise` and `rotate_clockwise`. The good news is that the logic behind one method is the symmetrical inverse of the other, so we only need to analyze one method in depth. We'll do this with the `rotate_counterclockwise` method.

We pass the arguments `a` and `b` into the `rotate_counterclockwise` method, representing the A and B nodes from this diagram:

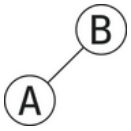


Alternatively, we could have passed one of the nodes in, such as A, and deduce that B is the right child of A since that is always the case when we perform a counterclockwise rotation. However, I prefer setting up the method signature in such a way that makes it clear which two nodes we'll be rotating.

For a moment, let's skip to the end of the method, where the essence of the rotation takes place:

```
b.left_child = a
a.parent = b
```

Before the rotation, B was the right child of A. With this snippet, we transform A into the left child of B, which yields:



The rest of our method deals with the crossover node and a couple of other details. So let's jump back to the top of the code, which handles all of that:

```
a.right_child = b.left_child

if b.left_child:
    a.right_child.parent = a
```

First, we update A's right child. Previously, A's right child was B. However, now that A no longer has B as a child, we need to update this.

Now, if B has a left child, this child will be the crossover node. Accordingly, we turn the crossover node into A's right child. On the other hand, if B does not have a left child, this means that its child is `None`, so now A's right child will also be `None`.

We then check to see if B actually did have a left child, which again is the crossover node. If this is the case, we make sure to set the crossover node's parent to now be A.

Next up, we redefine who B's parent is. Again, B's parent *used* to be A, but our rotation is going to turn B into A's parent. That leaves us hanging with the issue of who B's parent will be. Here's the code that deals with this:

```
b.parent = a.parent
if not b.parent:
    self.root = b
elif b.parent.left_child == a:
    b.parent.left_child = b
else:  # Node A was a right child
    b.parent.right_child = b
```

Because B is taking over A's spot, we need to explicitly turn B into the child of the node that was A's parent. Whenever you do something like this, you need to perform two distinct steps:

1. Declare B's `parent` to be what was A's parent
2. Declare within A's parent node that it now has a child B

If A was its parent's left child, then we need to make B its new parent's left child. And if A was its parent *right* child, then we need to make B its new parent's right child.

If it turns out that B's new parent is `None`, that means that B is in the root position and we need to declare that the tree's `self.root` is now B.

The `rotate_clockwise` method, as mentioned, has the exact same logic as `rotate_counterclockwise`, except that everything is the mirror opposite.

We're finally ready to explore a red-black tree's most important operation: insertions.

Red-Black Tree Insertion

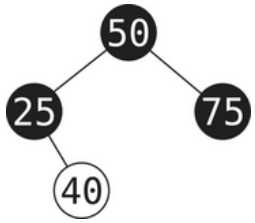
Inserting a new node into a red-black tree is considerably more complicated than inserting a node into a classic BST. We can't just plop the node into its correct spot; we also need to ensure that the tree follows the Red-Black Rules.

The first thing to know about red-black tree insertion is that the node we insert always starts out colored *red*. The reason for this will become apparent soon.

The second thing to know is that we insert a new node into a red-black tree in up to two phases. The first phase is virtually identical to a classic BST insertion. That is, we start at the root and move down through the tree searching for the correct spot to insert the new node based on its value. This follows the BST rule previously mentioned: namely, that a node's left descendants must have smaller values and a node's right descendants must all have greater values. I covered this process back in Volume 1, Chapter 15, but we'll see code for this again shortly.

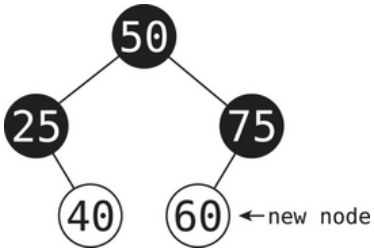
Once we find the correct spot, we attach the node to the tree by making it a child of another node. Once this first phase is complete, the inserted node will be a leaf node.

Now, if the newly inserted red node ends up having a black parent, we're done, and there's no need to move on to a second phase. Let's look at a quick example of this. Say that we have a red-black tree like the following:



If we want to insert a node with the value of 60, we start by comparing 60 to the root value. In this case, the root is 50.

Because 60 is greater than 50, we look to the root's right child, which in this case is 75. Because our new node, 60, is less than 75, we look to the 75's left child. However, the 75 has no left child. As such, we insert our new node as the 75's left child:



As with all newly inserted nodes, we've colored the 60 node red. Currently, there are no violations of the Red-Black Rules, so our insertion is complete. If, however, our new red node ends up with a parent that is *also* red, we will have violated the Red Enemies Rule, so we need to move on to a second phase, in which we "fix" the tree. This "fixing phase" modifies the tree in all sorts of ways, including changing nodes' colors and performing rotations.

Before we move on to the fixing phase, though, let's implement the first phase of insertion.

Code Implementation: Red-Black Tree Insertion (First Phase)

The following is our `insert` method:

```
    def insert(self, value):
        new_node = rbt_node.Node(value, "red")

        if not self.root:
            self.root = new_node
            return

        current_node = self.root
```

```
        while current_node:

            if value < current_node.value:
                if not current_node.left_child:
                    current_node.left_child = new_node
                    new_node.parent = current_node

                current_node = current_node.left_child

            elif value > current_node.value:
                if not current_node.right_child:
                    current_node.right_child = new_node
                    new_node.parent = current_node

                current_node = current_node.right_child

            else:  # value is already inside tree
                break

        self.fix_insert(new_node)
```

We call the method and pass a `value` into it. The first thing the method does is create a red node to encapsulate this `value`:

```
    new_node = rbt_node.Node(value, "red")
```

All the remaining code, save for the final line, is a classic BST insertion. Now, in Volume 1, we implemented this using recursion, which led to some concise and elegant code. Here, however, we've implemented BST insertion using iteration rather than recursion.

While we could have used recursion here as well, much of the code that is to come is more easily understood using iteration. To keep a consistent code style throughout this chapter, I've used iteration for this method as well, even though the code ends up being a little longer.

Let's briefly walk through the method's remaining code.

After creating the new red node, we declare it to be the tree's root if the tree doesn't yet have any nodes other than this one. We then set a variable `current_node` to point at the root. Next, with a `while` loop, we compare our new node's value to the value of the `current_node` and keep moving down through the tree via left or right children until we hit the spot where the new node should live. Once we've identified that spot, we insert the new node by making it a child of some existing leaf node.

In this implementation, we (arbitrarily) do not allow for duplicate values in our tree and `break` from the loop if we find that the value is already present in the tree.

This brings us to the method's final line of code:

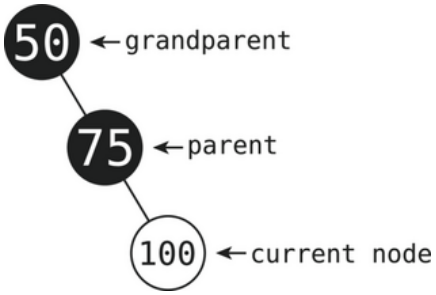
```
self.fix_insert(new_node)
```

This method, to be implemented soon, will check whether our new node violates the Red Enemies Rule, and if it does, modify the tree so that the tree becomes valid again. The `fix_insert` method is the core of how red-black trees work, and performs the fascinating magic of keeping the tree balanced. Let's see how it works.

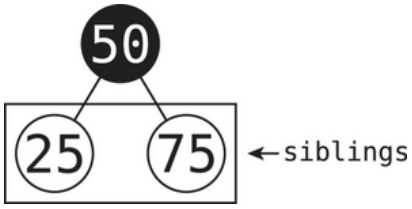
The Main Balancing Act: Fixing the Red-Black Tree

The fixing phase has lots of details, many of which will at first seem arbitrary and needlessly complex. To make things easier to comprehend, I begin with a high-level overview of how the fixing phase works. I'll first focus on what the steps are and later return to explain *why* the steps do what they do.

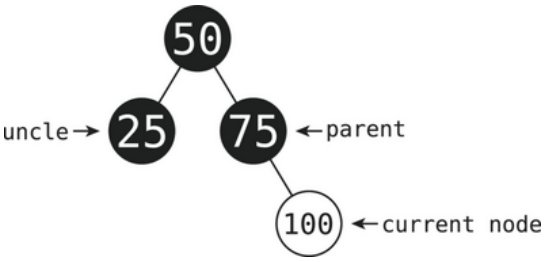
First, remember that a node can have a left child and a right child and that a node is considered a parent to its children. Keeping with the familial jargon, a node can have a *grandparent*, which is the node's parent's parent. In the following image, the 100 node's parent is the 75, and its grandparent is the 50:



Nodes are considered *siblings* if they share the same parent. In the following image, the 25 and 75 are siblings since they are both children of the 50:



Finally, a node can also have an *uncle* or, if you prefer, an *aunt*. These terms can be used synonymously, but we'll use the term "uncle" only because most of the literature does. A node's uncle is its *parent's sibling*. For example, in the image that follows, the 25 is the 100's uncle since the 25 is a sibling to the 75, which in turn is the 100's parent:



With these definitions in place, let's dig into the entire insertion algorithm, including its fixing phase. We'll first describe this piece by piece, and then afterwards make a clean list of all the algorithm's steps.

Remember, each time we insert a new node, we color it red. The color may eventually change, but when we first insert it, it's red. Then, we use the rules of a BST to move down the tree and determine where this shiny new red node should go. Once we find the right spot, we plug it in.

If the new node's parent is black, we're done. This is because we know that we could not have possibly broken any of the Red-Black Rules by inserting this node. We couldn't have broken the Black Height Rule because inserting a red node will not change the number of black nodes in a path. And we couldn't have broken the Red Enemies Rule because our new red node has a black parent; we didn't create a situation where a parent and child are both red.

If, however, our new node's parent is *red*, we've broken the Red Enemies Rule, and the fun begins. It's time for the fixing phase!

We set a variable called `current_node` to point to our newly inserted red node. Eventually, `current_node` will point to other nodes further up the tree, but at first, it points to our new node.

We then begin a loop which I'll refer to as the "fixing phase loop." The fixing phase loop lasts as long as the `current_node` and its parent are both red.

Within this loop, we look for one of three possible cases:

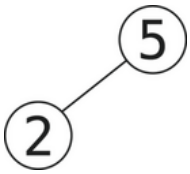
1. The `current_node`'s parent is the tree's root. We'll call this the "Root-Parent Case."
2. The `current_node`'s uncle is red. We'll dub this the "Red-Uncle Case."
3. The `current_node` doesn't have an uncle or its uncle is black. Even though these are technically *two* different scenarios, we'll consider them a single case since we perform the same actions for both. I'll call this the "Missing-Or-Black-Uncle Case."

Depending on what our case is, we'll perform a different series of actions that will help fix the tree. Some of these actions may seem arbitrary, but I'll try to uncover at least some of the rationale for each one.

The Root-Parent Case

Imagine that we only have a single node in our tree. By definition, this node is the tree's root. It will also be red since at the time that we inserted it, we colored it red, as we do with all insertions:

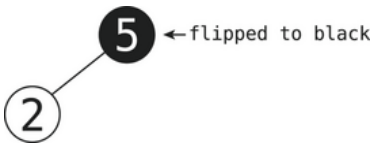
Isn't it cute? Okay, let's now insert a 2 into this tree:



Wow, we're only two nodes in, and we've already managed to break the Red Enemies Rule. This triggers our loop, which analyzes the `current_node` to see which of the three cases it falls under.

This case is the Root-Parent Case since the `current_node`, which is the 2, has the root as its parent. In truth, it should also qualify as the Missing-Uncle Case since the `current_node` has no uncle, but the fact that its parent is the root takes precedence.

Luckily, there's only one action we need to take in the Root-Parent Case. And that is, we color the root black:

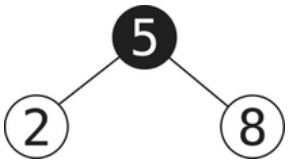


This single action solves the red enemies problem since we no longer have a parent and child who are both red. At the same time, the fact that we colored a node black didn't violate the Black Height Rule either. This is because when we color the *root* black, *all* the paths in the tree increase their black height by one. Even in large trees, coloring the root black can't possibly add a black node to only one path and not another. After all, the root belongs to all paths!

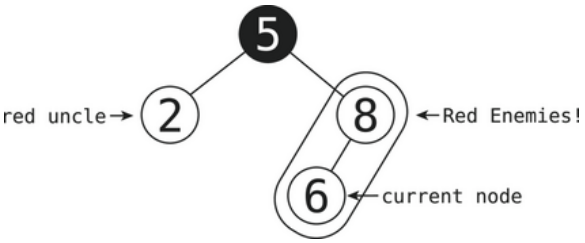
That was simple enough. Let's move on to the Red-Uncle Case.

The Red-Uncle Case

Take a look at the following tree:



If we insert a 6, our tree will initially look like this:

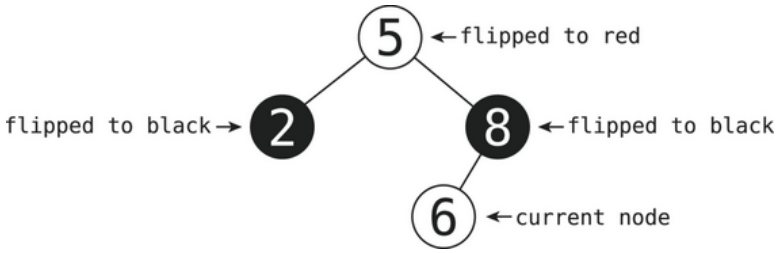


Our newly inserted node and its parent, the 8, are both red and are therefore in violation of the Red Enemies Rule. This triggers the fixing phase loop.

This scenario represents a Red-Uncle Case since the `current_node`'s uncle, the 2, is red. Here is the set of actions we take in a Red-Uncle Case:

1. We color the `current_node`'s parent black.
2. We color the `current_node`'s uncle black.
3. We color the `current_node`'s grandparent red.
4. We set the `current_node` variable to point to the `current_node`'s grandparent, and go back to the beginning of the fixing phase loop. (The loop will proceed again if the new `current_node` and its parent form a new red enemies violation.)

An image of what our tree looks like after the first three steps is shown.



We can see that the Red Enemies Rule has been resolved, for there are no longer any red nodes touching each other. At the same time, we also didn't mess up the Black Height Rule; each path has the same black height of 1.

This set of color flips seems random and arbitrary at first. Yes, flipping the parent and uncle black while flipping the grandparent red seems to do the job, but why?

To make more sense of this, I like to think about the grandparent as bequeathing its black color to its two children as an inheritance. That is, the grandparent turns red and gives up its black color to its children.

Now, this immediately resolves the red enemies violation because one of these children that inherited black color is the parent of our `current_node`. In our example, this was the 8. This 8 was a red enemy, but because it turned black, it can happily be a parent to its red child (the `current_node` 6).

At the same time, when the grandparent bequeaths its black color to its two children, it automatically maintains the Black Height Rule. In our example, the grandparent, the 5, has two paths descending from it. Because it gives up its black color to *both* paths, these paths will both increase their black height equally.

Therefore, this set of color flips solves the Red Enemies Rule while also maintaining the Black Height Rule.

Repeating the Fixing Phase Loop

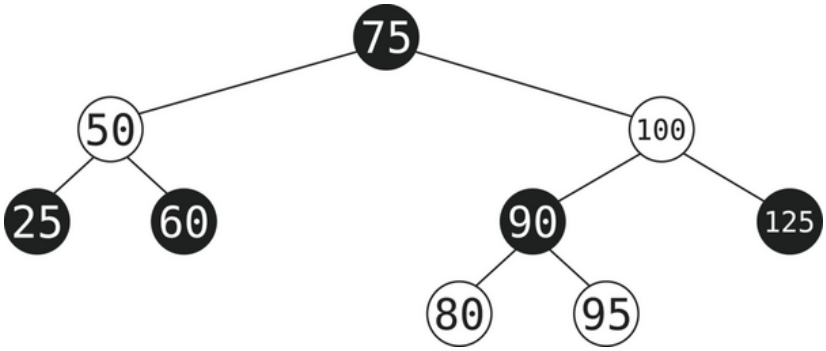
Now, after we perform all the color flips, we update the variable `current_node` to point to the grandparent of the newly inserted node. In our

example, this is the 5. We then jump back to the top of our fixing phase loop, which will continue again if the `current_node` and its parent have become red enemies as a result of our color flips.

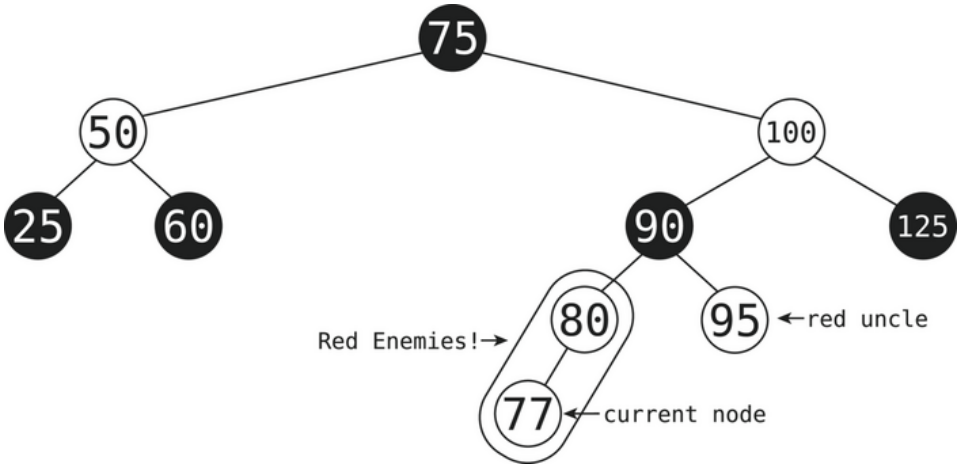
However, in our previous example, the loop terminates without repeating. This is because the `current_node` happens to be the root node, which, by definition, doesn't have a parent in the first place, so it can't have a parent who is also red. And so, we're done with our entire fixing phase, and the tree is once again in compliance with the Red-Black Rules.

Let's look at an example of where the fixing phase loop would run more than once.

Say that we have this tree:

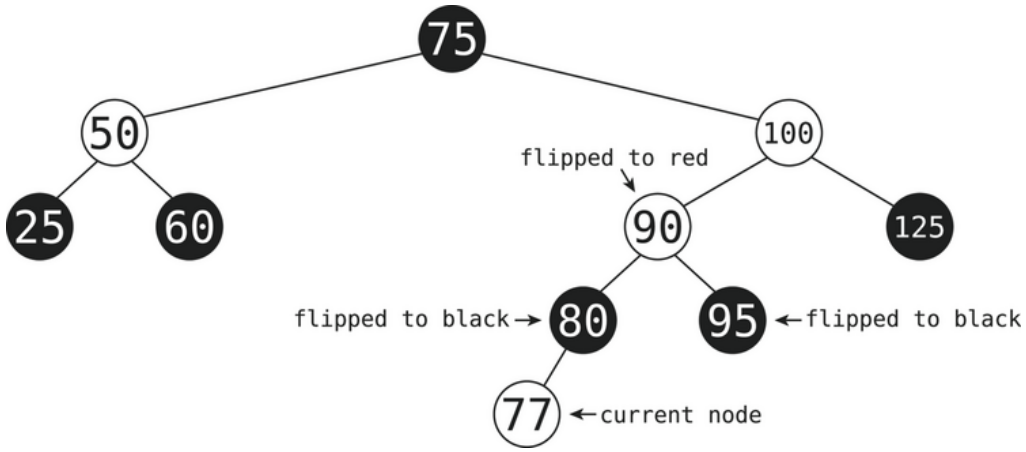


If we insert a 77, we get:

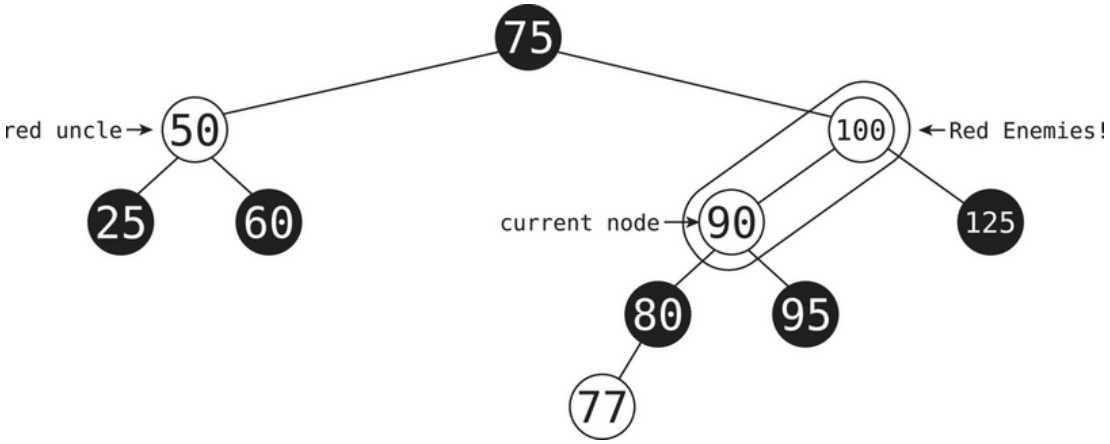


The 77 is red enemies with the 80, so our fixing phase loop begins.

This is the Red-Uncle Case since the 77's uncle, 95, is red. And so, we begin with the color flips. That is, we flip the grandparent red, and the grandparent's children (the `current_node`'s parent and uncle) black:

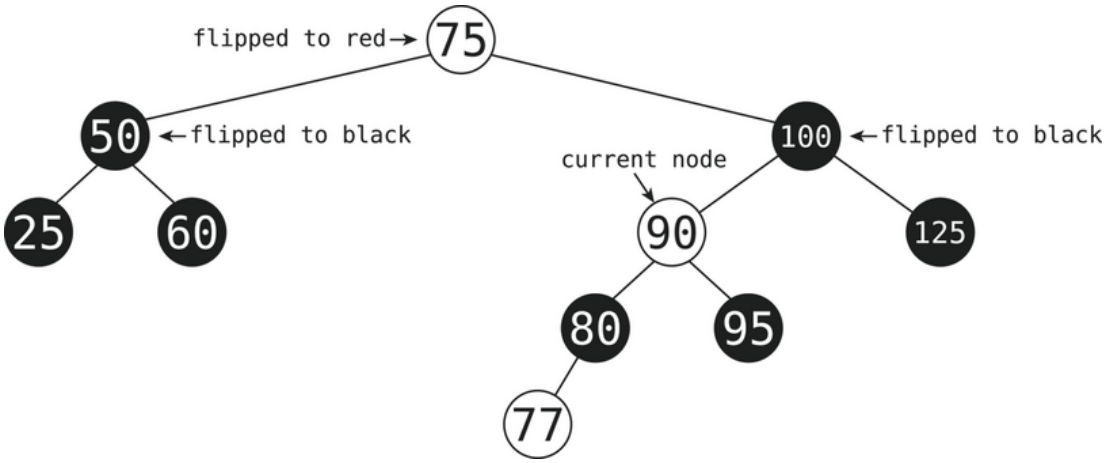


Before beginning the next round of the loop, we update the `current_node` to point to the grandparent, which in this case is the 90:



We check to see if the `current_node` and its parent are both red, and lo and behold, they are! It turns out that by resolving the red enemies violation of the 77 and the 80, we introduced a new red enemies violation with the 90 and 100.

Now, this is also a Red-Uncle Case since the 90's uncle, 50, is red. This means that we need to perform our color flips again, namely, turning the grandparent red and the parent and uncle black:

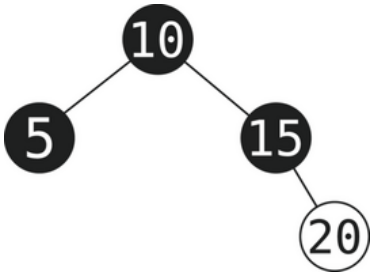


We gear up for the loop's next round by turning the 90's grandparent into the `current_node`. In this case, the `current_node` is now the root. And because the root and its parent aren't both red (since the root doesn't even have a parent), we're done.

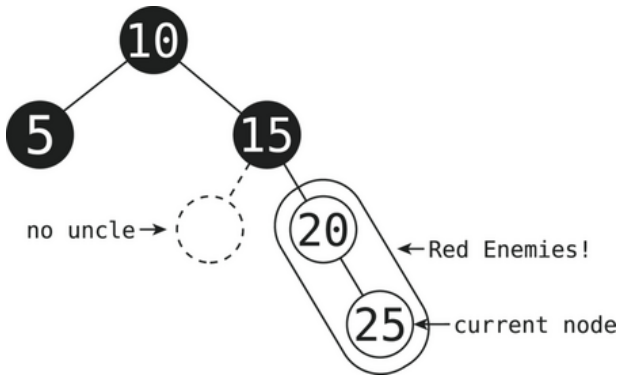
We've successfully covered the first two cases of the fixing phase: the Root-Parent Case and the Red-Uncle Case. This brings us to our final case.

The Missing-Or-Black-Uncle Case

Take a look at the section of a red-black tree shown.



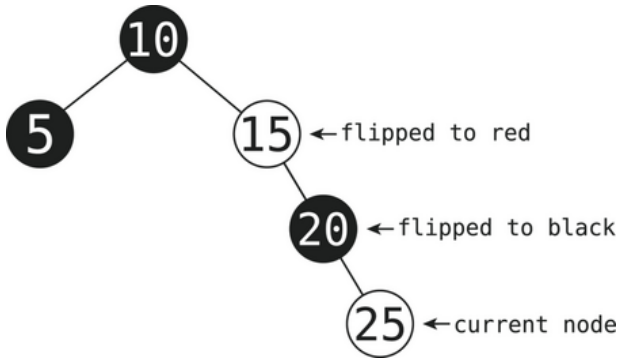
If we insert a 25, we get a red enemies violation:



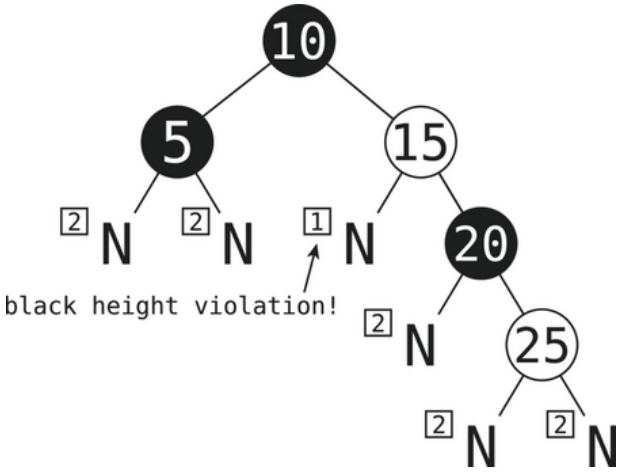
To resolve this violation, we begin our loop. Now, this is a Missing-Uncle Case since our `current_node` 25 has no uncle. (That is, its parent, 20, has no sibling.)

The Wrong Solution

Let's first explore why we can't use the same solution from the Red-Uncle Case. Again, the technique in the Red-Uncle Case is to flip the grandparent red and have the grandparent bequeath its black color to its children. This potentially fixes things because one of the grandparent's children is the `current_node`'s parent. And so, if the `current_node`'s parent becomes black, we resolve the red enemies violation between the `current_node` and its parent. In this case, the grandparent has only one child, so our tree will become this:



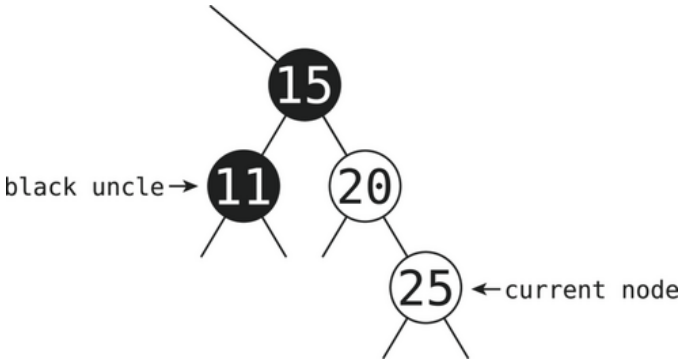
This tree appears to stick to all the Red-Black Rules, as there are no red enemies and each path seems to have the same black height. But, alas, this is not true. Remember the phantom nodes? We need to measure the black height of each path, keeping the phantom nodes in mind, as shown in the image.



While most of the paths have a black height of 2, the path coming off the phantom left child of the 15 has a black height of 1, so our tree would be in violation of the Black Height Rule.

So, the trick of having a grandparent bequeath its black color to its children only works when it can turn *both* of its children black. If it turns only one of its children black, it causes its descendant paths to have differing black heights.

This is also true in a Black-Uncle Case. Let's look at such a case:



We're only focusing on a piece of a larger tree since a Black-Uncle Case happens to never occur at the bottom of a tree. It can only happen during our loop as we work our way up through the tree.

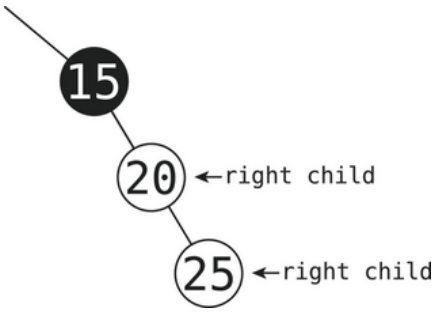
In any case, the 25's "Uncle 11" is black. If we were to try to resolve this by having "Grandma 15" bequeath her black color to both of her children, we'd be increasing the black height of her right-child path by one, while the

black height of her left-child path would remain the same because the left child, 11, is *already black*. So we're increasing the black height of the right path by one, while we're not at all increasing the black height of the left path. This, in turn, results in a black height violation.

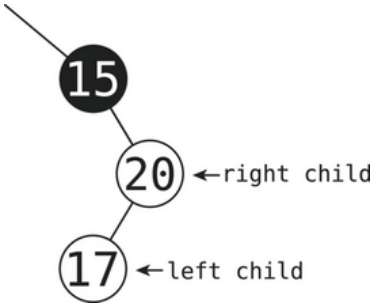
The Right Solution

It turns out that the technique used to resolve the Red-Uncle Case will not help for our case. Instead, for a Missing-Or-Black-Uncle Case, we actually *do* perform the aforementioned color flips. However, we *also* need to perform a rotation. We will perform either *one or two* rotations depending on the formation of the `current_node` and its parent and grandparent. Brace yourself, as the Missing-Or-Black-Uncle Case is subdivided into two further *sub*cases. (Evil laugh. Just kidding; it'll be fine.)

One subcase is where the `current_node` and its parent have the same orientation. That is, they are either both a right child or both a left child of their respective parents. For example:



Both the 25 (which is the `current_node`) and the 20 are right children of their respective parents. However, the other subcase is when the `current_node` and its parent have *opposite* orientations, like this:



The 17 (which is the `current_node`) is its parent's *left* child, while the 20 is its parent's *right* child.

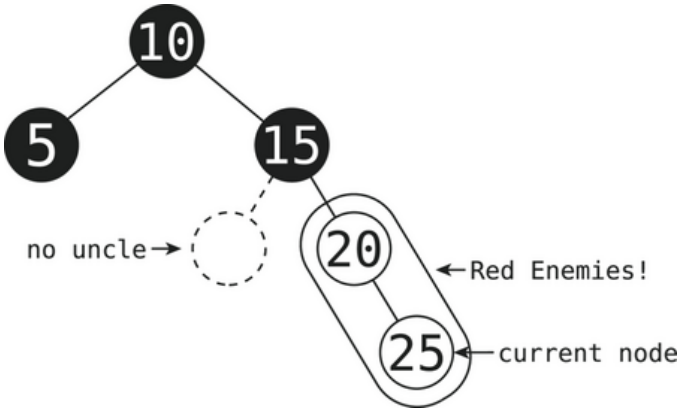
Let's take a look at how to deal with both subcases.

Subcase 1: Same Orientation

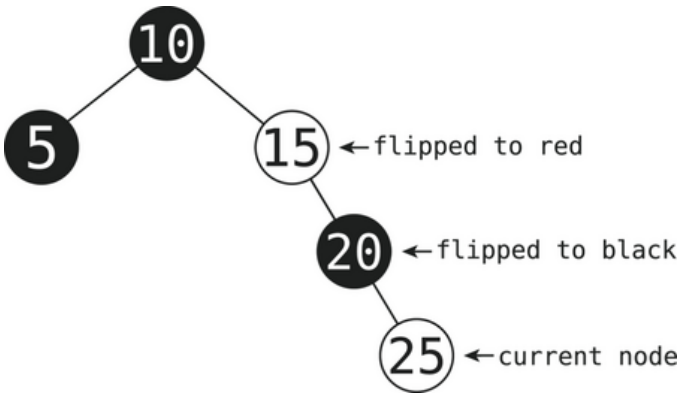
If the `current_node` and its parent have the same orientation, we only need to perform a single rotation plus a couple of color flips. Here's the precise algorithm:

1. Flip the `current_node`'s parent black.
2. Flip the `current_node`'s grandparent red.
3. Perform a rotation between the `current_node`'s parent and grandparent. (Whether it's a clockwise or counterclockwise rotation depends on whether the parent is a left or right child of the grandparent.) And then you're done.

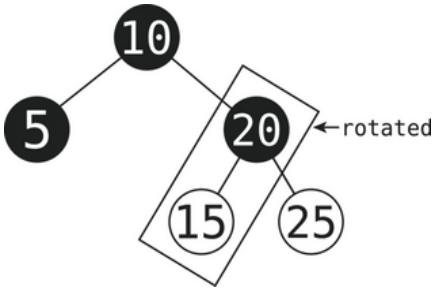
Let's take a look at this in action. Here's our current Missing-Uncle Case:



We flip the `current_node`'s parent black and grandparent red:



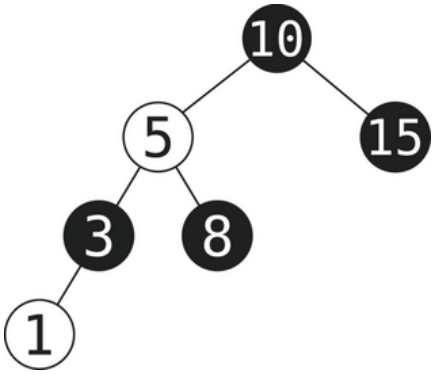
We then perform a rotation of the parent and grandparent, that is, the 20 and the 15. In this case, because the parent is the grandparent's *right* child, we perform a counterclockwise rotation:



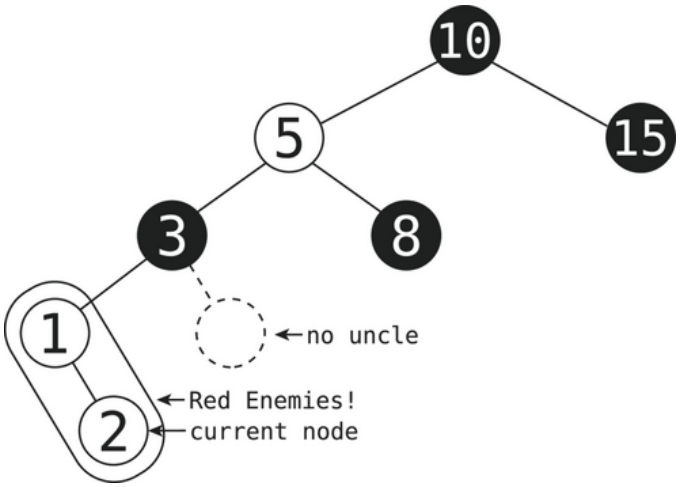
And we're done! Our tree is now completely fixed.

Subcase 2: Opposite Orientations

When you encounter the other subcase, in which the `current_node` and its parent have opposite orientations, you need to perform one extra rotation before moving on to the same set of steps from the previous subcase. Take the following example:



If we insert a 2, we get:



The 2 and 1 form a red enemies violation. The `current_node` (the 2) has no uncle, so this is a Missing-Uncle Case. In particular, it's the subcase in which the `current_node` and its parent don't have the same orientation. That is, the 2 is a right child, but the 1 is a left child. This means we're going to perform two rotations.

We'll see in the next section exactly *why* we need two rotations, but let's proceed with the algorithm details:

1. Perform a rotation between the `current_node` and its parent. Pay special attention to the fact that this rotation will transform the `current_node`'s *parent* into the `current_node`'s *child*.

2. Make this new child the `current_node`.

3. Flip the `current_node`'s parent black.

4. Flip the `current_node`'s grandparent red.

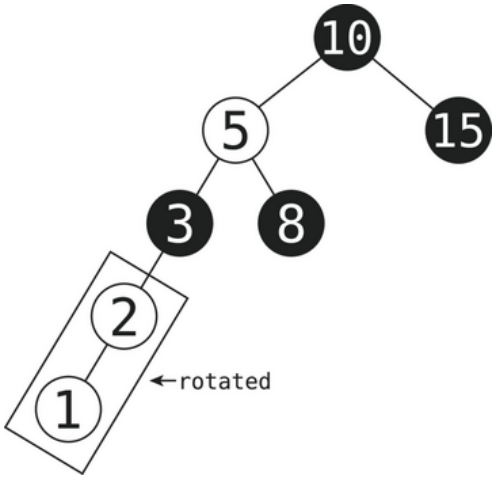
5. Perform a rotation between the `current_node`'s parent and grandparent, and then you're done.

Note that steps 3, 4, and 5 for this subcase are identical to the final three steps of the first subcase. But in this second subcase, we perform a couple

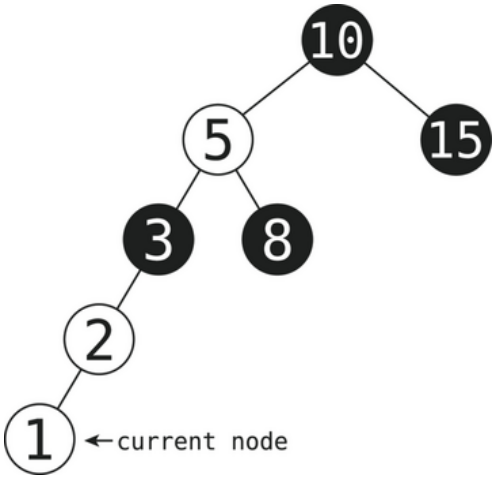
of extra steps first.

Returning to our example, we've inserted the 2 and created a red enemies violation with its parent. Our next step is to perform our first rotation, which rotates the `current_node` and its parent.

In our case, the 2 is the 1's right child, so we perform a counterclockwise rotation:



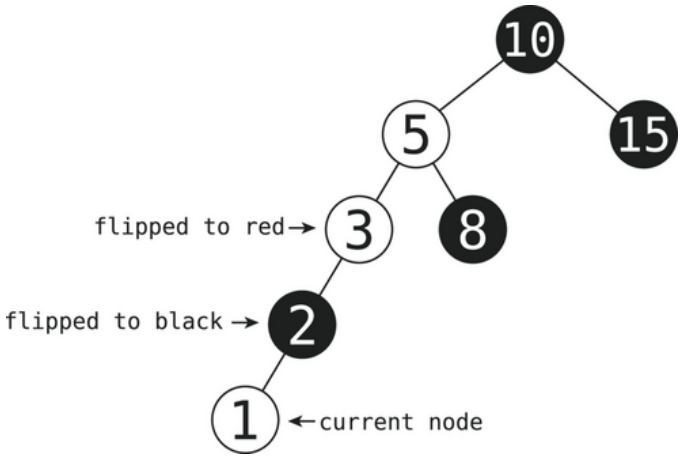
Next, we update the 1 to be the new `current_node`:



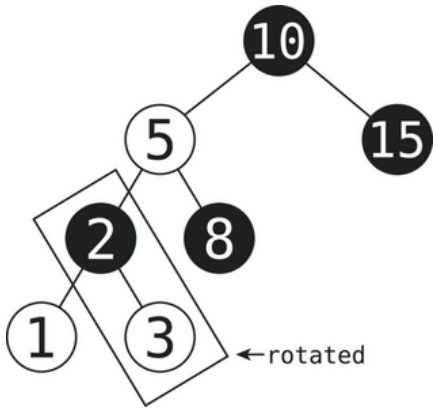
Updating the `current_node` in this way isn't a critical step, but doing so makes it so that the remaining steps of our algorithm can be described in exactly the same way as the steps of the algorithm for the previous subcase.

Again, Steps 3–5 of this algorithm are the same as Steps 1–3 of the first subcase's algorithm.

Next up, we flip the `current_node`'s parent black, and grandparent red:



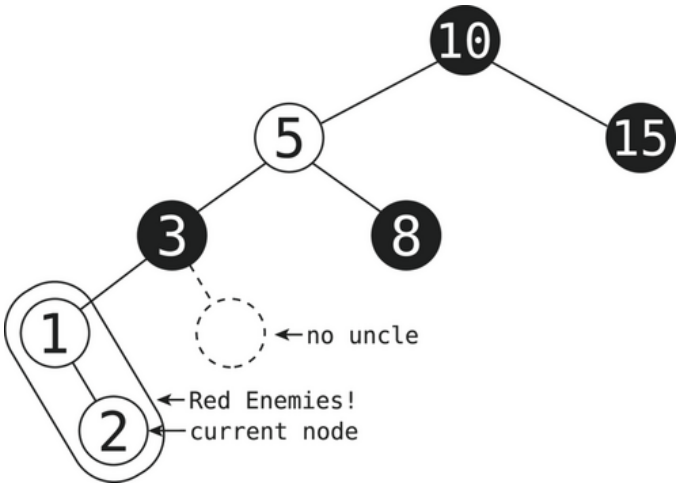
We now perform the second (and final) rotation. Specifically, we rotate `current_node`'s parent and grandparent. This will be a clockwise rotation because the parent is the grandparent's left child:



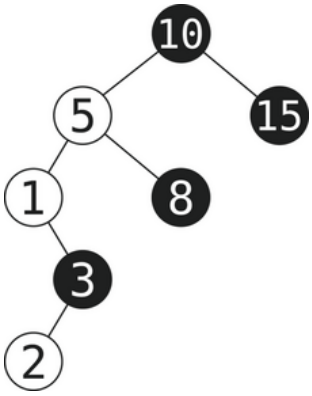
And our red-black tree is completely fixed!

Why the Double Rotation

To see why we needed two rotations for this last subcase, let's return to the moment where we first inserted the new node:

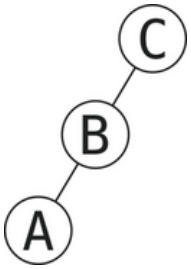


If we tried to jump right away to the final rotation, where we rotate the `current_node`'s parent and grandparent (the 1 and the 3), we get this:

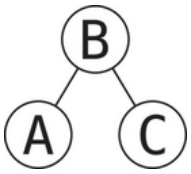


This step doesn't get us any closer to balancing the tree. Before this rotation, our tree was five levels deep, and after the rotation, it's still five levels deep.

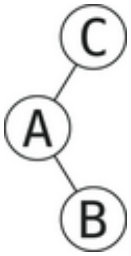
What this boils down to is this: when we have a node, parent, and grandparent that are all in the *same* orientation, rotating the parent and grandparent will balance that tree segment. For example, take this three-level tree segment:



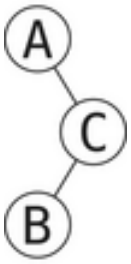
When we rotate the parent and grandparent, we turn it into the following perfectly balanced two-level segment:



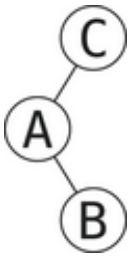
But, say we have *this* tree segment where the A and the B do *not* have the same orientation:



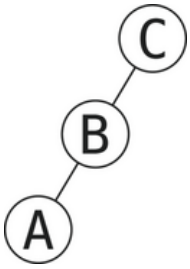
Rotating the parent and grandparent merely turns it into a mirror of itself without doing anything to balance it:



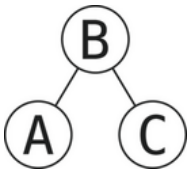
So instead, we do a *double* rotation. That is, if we have a segment like this:



We *first* rotate the bottom node and its parent (in this case, the B and A):



And *then* we rotate the new parent with the grandparent (the B and the C):



And so the tree becomes balanced.

With a double rotation, I like to think of the first rotation as being the *segment straightener*—it straightens out the segment so that both the bottom node and its parent are now of the same orientation. Once the segment is straightened, we perform the second rotation, which perfectly balances that segment.

Wow. We've gone through a lot of details. If your head isn't spinning a little bit, that makes one of us. Ready to put it all together?

The Grand Finale: The Complete Insertion Fixing Algorithm

I now present to you The Complete Insertion Fixing Algorithm. It may not be fun to read, but it's a great reference. Here goes:

1. Insert the new node in the correct spot in the tree (according to BST rules) and color the new node red. This new node is dubbed the `current_node`.

2. Begin a loop that lasts while the `current_node` and its parent are both red. (If they're not both red, skip to Step 17—you're done!)

3. If the parent is the root, color the root black and skip to Step 17—you're done.

4. Inspect the `current_node`'s uncle to see if it exists and what color it is.

5. If the uncle is not red, break out of the loop and skip to Step 11; this is the Missing-Or-Black-Uncle Case.

6. If the uncle is red, this is the Red-Uncle Case and proceed with the steps 7–11.

7. Color the `current_node`'s parent black.

8. Color the `current_node`'s uncle black.

9. Color the `current_node`'s grandparent red.

10. Make the grandparent of the `current_node` the new `current_node` and repeat the loop from Step 2.

11. Check if the `current_node` and its parent have the same orientation. If they do, skip to Step 14. If they do not have the same orientation, proceed with steps 12–17.

12. Perform a rotation between the `current_node` and its parent.

13. This rotation transformed the `current_node`'s parent into the `current_node`'s *child*. We make this new child the `current_node`.

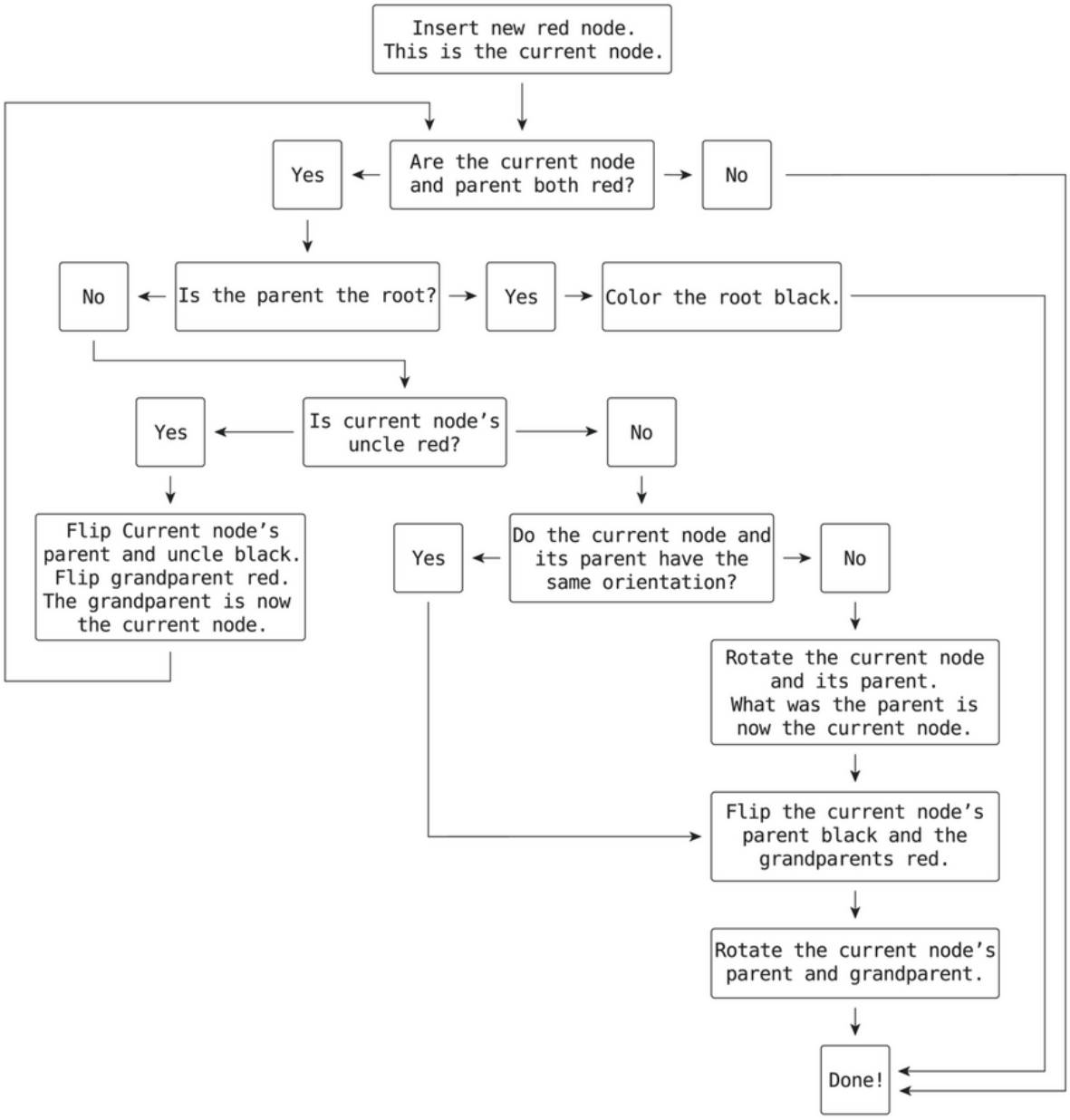
14. Flip the `current_node`'s parent black.

15. Flip the `current_node`'s grandparent red.

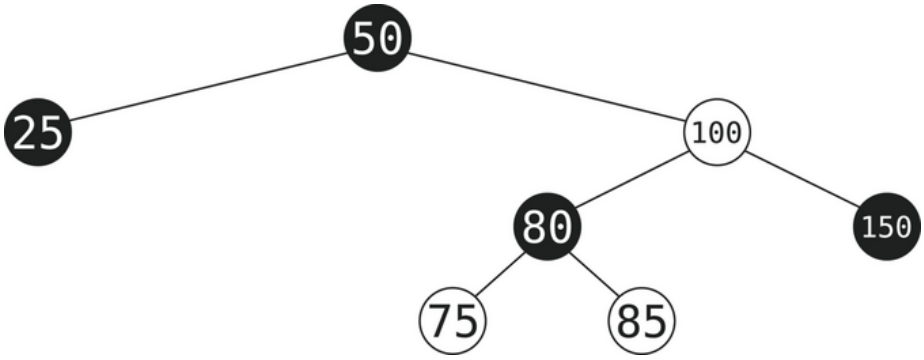
16. Perform a rotation between the `current_node`'s parent and grandparent.

17. Top with lemon or your favorite garnish. Serves 6.

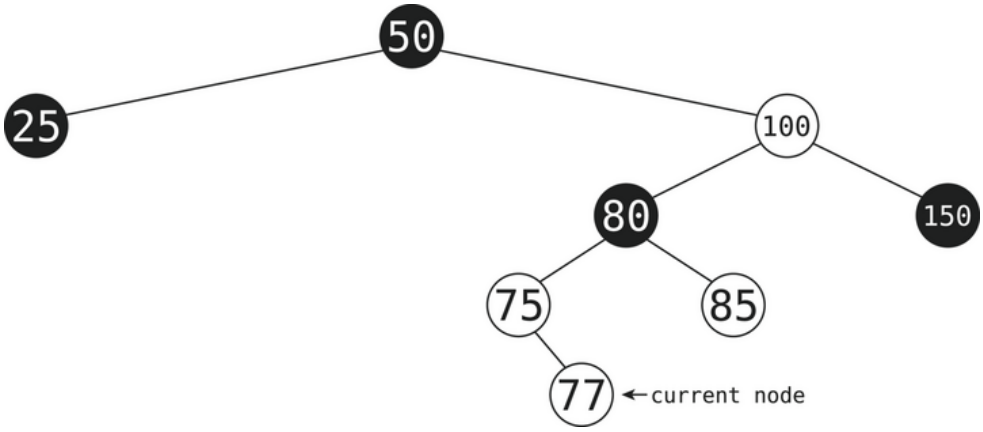
If you're a flowchart person, look—I made something for you. The flowchart shown depicts these steps in visual form.



Let's walk through one final example. The step numbers that follow correspond to the step numbers in our list of steps. Take the following tree:



Step 1: We insert a 77, as shown in the tree.

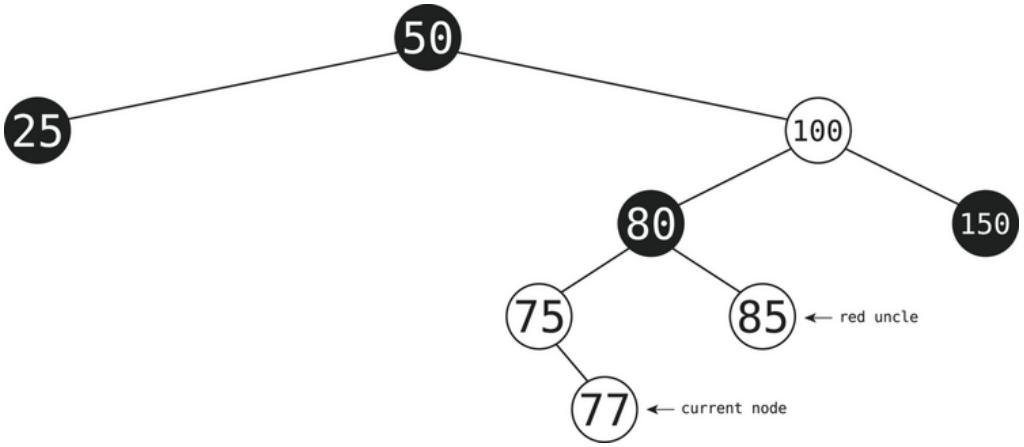


The 77 is the `current_node`. Both it and its parent, the 75, are red.

Step 2: We begin a loop that will last as long as the `current_node` and its parent are both red.

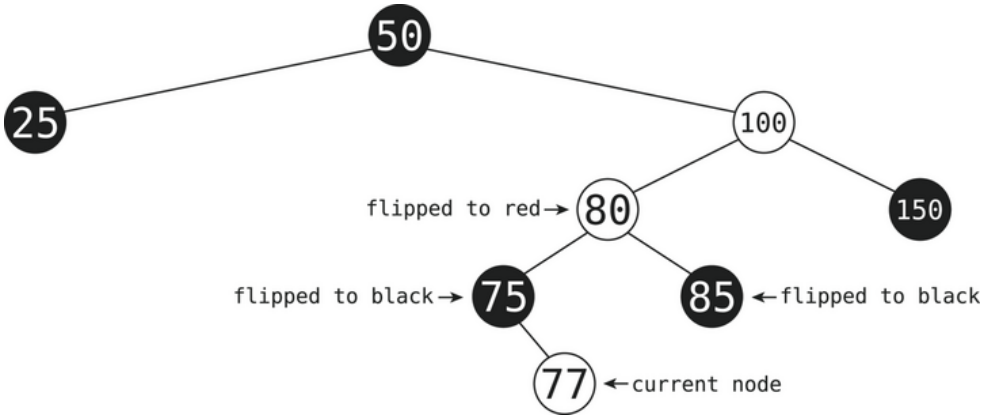
Step 3: The parent is not the root, so we'll move on to the next step.

Step 4: Inspect the `current_node`'s uncle to see its color:

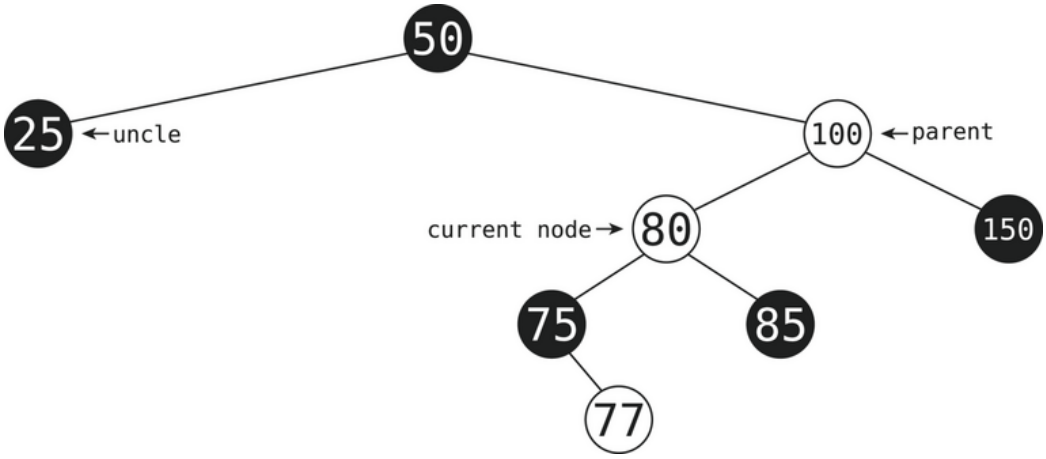


Steps 5 and 6: “Uncle 85” happens to be red. This is a Red-Uncle Case, and we therefore proceed through the loop.

Steps 7, 8, and 9: We flip the `current_node`’s parent and uncle black, and flip the grandparent red:



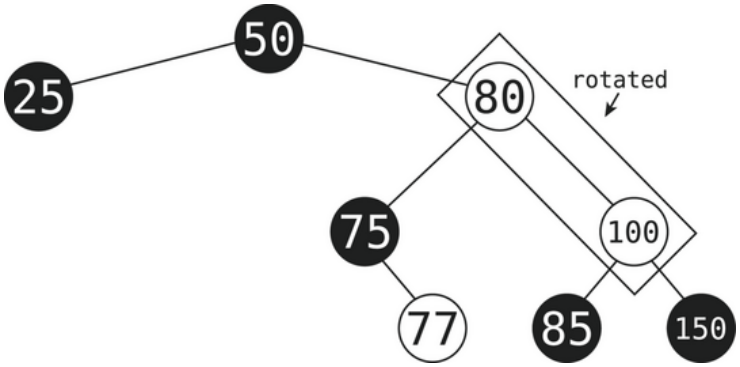
Step 10: We turn “Grandpa 80” into the `current_node`:



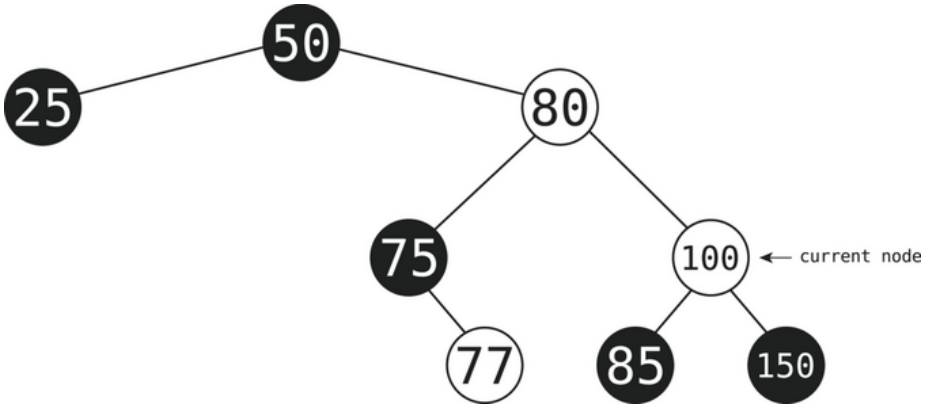
We go back to Step 2 to begin the loop again. The `current_node` and its parent are both red, so we continue to Step 3, which inspects the uncle. “Uncle 25” is black, which brings us to Step 5, which tells us to break out of the loop and skip to Step 11.

Step 11: We check whether the `current_node` and its parent have the same orientation. They do not, so we continue to Step 12.

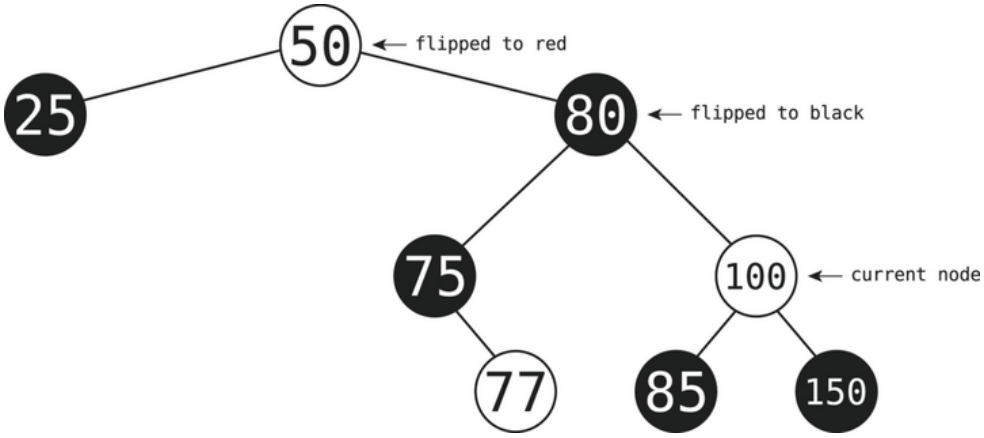
Step 12: We rotate clockwise the `current_node`, 80, and its parent:



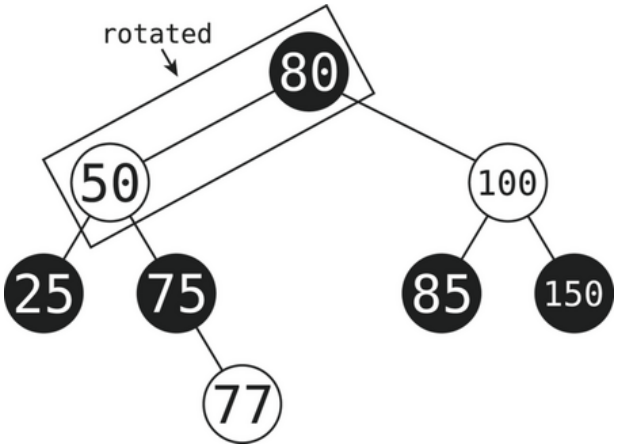
Step 13: The 100, which is now a child of the 80, becomes the new `current_node`:



Steps 14 and 15: Flip the `current_node`’s parent black, and grandparent red:



Step 16: Rotate the `current_node`'s parent and grandparent counterclockwise:



And WE. ARE. DONE.

Ready for some code?

Code Implementation: Red-Black Tree Insertion (Fixing Phase)

For this code implementation, I added comments within the code to connect each line to its corresponding step from *The Grand Finale: The Complete Insertion Fixing Algorithm*. Instead of boring you with a long-winded code walk-through, I'll let you match up each line of code to our list of steps.

Note that the main method is the `fix_insert` method, but it does rely on a series of helper methods:

```
def fix_insert(self, current_node):
    # Step 1 was accomplished by the `insert` method
```

```
    # which calls this fix_insert method
    while self.has_red_parent(current_node):           # Step 2
        if current_node.parent == self.root:           # Step 3
            self.root.color = "black"
            return

        uncle = self.find_uncle(current_node)          # Step 4
        if uncle and uncle.color == "red":             # Step 6
            current_node.parent.color = "black"        # Step 7
            uncle.color = "black"                      # Step 8
            current_node.parent.parent.color = "red"   # Step 9
            current_node = current_node.parent.parent  # Step 10
        else:  # uncle is None or black                  Step 5
            break

    if self.has_red_parent(current_node):
        # The next line calls the straighten_segment method
        # which accomplishes Steps 11, 12, and 13
        current_node = self.straighten_segment(current_node,
                                               current_node.parent)
        current_node.parent.color = "black"               # Step 14
        current_node.parent.parent.color = "red"          # Step 15
        self.perform_final_rotation(current_node.parent)  # Step 16

def has_red_parent(self, node):
     return node.parent and node.parent.color == "red"

def parent_is_root(self, node):
    return node.parent == self.root

def is_a_left_child(self, node):
    return node == node.parent.left_child

def is_a_right_child(self, node):
    return node == node.parent.right_child

def find_uncle(self, node):
    if self.is_a_left_child(node.parent):
        return node.parent.parent.right_child
    else:  # parent is a right child
        return node.parent.parent.left_child

def straighten_segment(self, node, parent):
```

```
        former_parent = parent
        if self.is_a_left_child(parent) and self.is_a_right_child(node):
            self.rotate_counterclockwise(parent, node)
            return former_parent
        elif self.is_a_right_child(parent) and self.is_a_left_child(node):
            self.rotate_clockwise(parent, node)
            return former_parent
        else:  # no need to straighten the segment
            return node

    def perform_final_rotation(self, node):
        # If the node's parent is a left child of its OWN parent:
        if self.is_a_left_child(node):
            self.rotate_clockwise(node.parent, node)
        else:  # if the parent is instead a right child:
            self.rotate_counterclockwise(node.parent, node)
```

The Efficiency of Red-Black Trees

Because a red-black tree is a type of binary search tree, the search operation is O(log N), as is true for all BSTs. As I discussed in Volume 1, Chapter 15, this is because there are log N levels in a BST, and at most, we will search one node from each level in the BST until we find the value we're searching for. I also discussed there that insertion for a regular BST is O(log N), as we must first perform a search to find the appropriate spot to insert the new node, and the actual insertion is one step.

It turns out that insertion in a red-black tree is also O(log N). Now, there are certainly more steps involved with insertion into a red-black tree than there are with a regular BST. However, it'll add up to roughly 2*log N steps at most. Here's why.

You learned that there are two phases for insertion into a red-black tree: insertion and fixing. The insertion phase is identical to insertion in a BST, which takes log N steps. The fixing phase starts with the newly inserted node and works its way up the tree, performing color flips and rotations. Just as we took log N steps to get from the top of the tree to the bottom of the tree, it takes at most another log N steps to move from the bottom of the tree back up to the top of the tree.

In total, this is 2*log N steps, which boils down to O(log N).

It turns out that red-black trees are a win-win. They maintain the tree's balance while keeping its operations swift and efficient.

Red-Black Tree Deletion

After a lot of hard work, you've seen how a red-black tree's insertion operation works. But wait, there's more! Don't forget that there's also an algorithm for *deleting* a node from a red-black tree. As complex as insertion was, deletion involves even *more* steps and *more* subcases. However, this chapter is already too long, so you've been saved by the bell.

For completeness, you can find a description of the deletion algorithm on the book's web page.[5] So, the choice is yours. If you're ready to move on from red-black trees, no problem. But if you need to learn this stuff for a college exam, the material is right there for you. (And good luck!)

In any case, I do recommend glancing through the deletion algorithm if only to appreciate its complexity because in the next chapter, we are going to contrast the complexity of red-black trees with the simplicity of another self-balancing tree called a *treap*.

Wrapping Up

A red-black tree, while complex, is a dependable data structure that ensures that search, insertion, and deletion remain speedy while also maintaining the order of its values. The key word here is "ensure." While a regular, good ol' BST *aims* to have a speedy efficiency, it can't absolutely *ensure* that speeds don't degrade, which can happen when values are inserted into the tree in order. A red-black tree, on the other hand, does ensure that its operations remain fast since the tree is guaranteed to be pretty well-balanced.

In fact, red-black trees are found under the hood of many applications, some of which we use every day. Some operating systems use them for various functions, including process scheduling, in which scheduled events need to be easily found but also kept in order of time. Some databases use red-black trees for database indexing, a concept we'll explore more fully in *B-Trees as Database Indexes*.

Also, I explained in Volume 1, Chapter 8, how collisions can occur within hash tables. There, I showed that one approach for handling this is to place all colliding values within an array or list using separate chaining. The disadvantage to this is that if we've encountered a value that collides with other values, we need to linearly search through all the colliding values until we find the one we want. As an alternative, some programming languages use red-black trees to store colliding values. This way, we can find any such value within O(log N) time, where N is the number of colliding values.

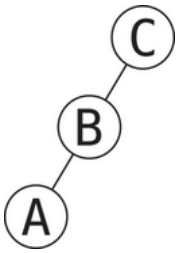
One of the biggest drawbacks of red-black trees, though, is their complexity. Sure, once they're implemented properly, they'll work as intended. But if you ever have to maintain them or improve them in some way, it could be a doozy to get it right.

In the next chapter, we're going to look at another type of self-balancing tree that works as well as a red-black tree, but is much, *much* simpler. And to achieve its performance and simplicity, it uses randomization. Once again, you'll see how the power of randomization can be used to achieve great things.

Exercises

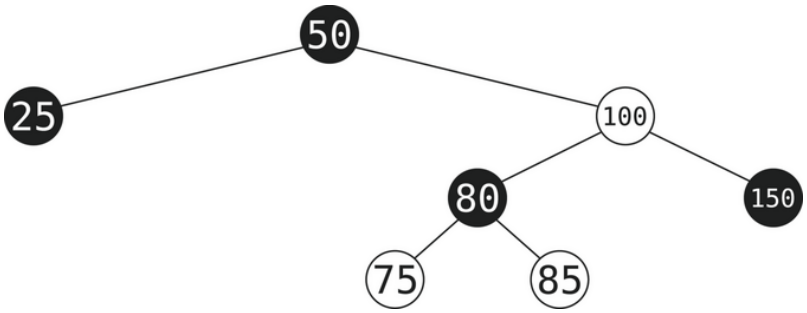
The following exercises provide you with the opportunity to practice with red-black trees. The solutions to these exercises are found in the section *Chapter 5*.

1. Here's a simple tree:



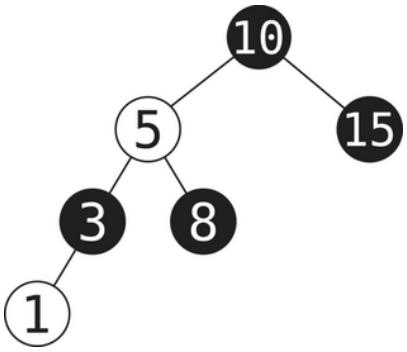
   It's not supposed to be a red-black tree, don't worry. However, we can still perform a rotation on it. Show what the tree will look like after you rotate the A and the B.

2. Here's a red-black tree:



   Rotate the 80 and the 100. Don't worry about violating the Red-Black Rules and having to fix up the tree after the rotation. Simply show what would happen if that one rotation happened to be made.

3. Here's another red-black tree:



What will the red-black tree look like if you insert a 30? This time, you must follow the entire red-black tree insertion algorithm.

4. This exercise is a continuation of the previous one. Say that after you insert the 30, you then also insert a 20. What will the red-black tree look like when all is said and done?

Footnotes

[5] https://pragprog.com/titles/jwpython2

Chapter 6

Randomized Treaps: Haphazardly Achieving Equilibrium

In the previous chapter, we looked at red-black trees and how they keep themselves balanced using a set of complex algorithms. We're now going to take a look at another type of self-balancing tree known as a *treap*. Treaps can maintain balance as effectively as a red-black tree, but with algorithms that are *way* simpler. And as you'll see, this simplicity is achieved through the power of … randomization.

Treaps

Like a red-black tree, a *treap* is a type of binary search tree (BST), but a treap follows a unique set of rules beyond the behavior of a traditional BST. The name "treap" is a combination of the words "tree" and "heap," and you'll see shortly what heaps have to do with treaps, other than the fact that they rhyme.

Treaps themselves come in various flavors, not all of which are self-balancing. Our focus in this chapter is on the *randomized treap* since this is the kind of treap that serves as a self-balancing tree. To save ink, though, I'll refer to randomized treaps simply as treaps since that's the only type of treap we'll be dealing with here.

Treaps vs. Red-Black Trees

Treaps and red-black trees share certain things in common:

1. They are both variants of BSTs. As a reminder, to be considered a BST, (1) a tree must have nodes with 0, 1, or 2 children; and (2) a node's left descendants must all have lower values than it, and a node's right descendants must all have greater values than it. For ease of reference, I'll call this the "BST Rule" throughout this chapter.

2. Red-black trees and treaps are both self-balancing trees. This means that even if we insert values in ascending order, the tree will not become a linked list, but instead will have a height along the lines of log N.

3. Treaps maintain balance using the same types of rotations red-black trees use. In *Rotations—Part 2*, I discussed the details of how clockwise and counterclockwise rotations work; treaps use these very same rotations.

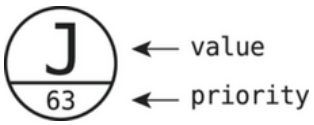
However, this is where the similarities between treaps and red-black trees end. While red-black trees maintain balance using the Red-Black Rules outlined in *The Red-Black Rules*, these rules do not exist in the world of treaps. In fact, treap nodes have no concept of color at all. Instead, treaps follow a simpler set of rules. And thank goodness for that.

Treap Nodes

Treap nodes are different than classic BST nodes in that, in addition to containing a value, each treap node also contains an extra piece of data. This extra data is called the *priority*, and comes in the form of a number. We'll see shortly what the purpose of the priority is.

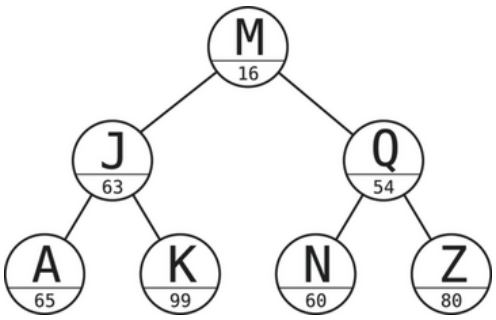
A node's priority is not at all correlated to the node's value. In fact, in a randomized treap, a node's priority is generated randomly.

Throughout this chapter, all of the treap diagrams will have nodes that contain a *value* that is a letter from the alphabet, and a *priority* that is a number between 1 and 100. (It's common to choose a priority that is a float that lies between 0 and 1, but I'm using the range of 1 to 100 to keep the diagrams simple.) Here's an example treap node:



Although in past chapters I used numerical values in tree examples, to easily distinguish between the value and the priority, I'm making values alphabetical characters in this chapter. Really, though, treap node values can be numbers (or any data type) as with any tree.

An example of a treap is shown.



Note that because a treap is a type of BST, it must follow the BST Rule, which is that the left descendants' values must be smaller than their ancestor, and the right descendants' values must be greater than their ancestor. Because we're using alphabet letters as values, whether a letter is smaller or greater than another letter depends on its placement within the alphabet. For example, the letter Q is considered "greater" than the letter H because Q appears later in the alphabet. The letter A is the "smallest" value in the entire alphabet, and the letter Z is the "greatest" value.

The Heap Rule

So far, we've seen that a special attribute of a treap is that its nodes contain priorities in addition to values. However, treaps have one other important attribute: a treap must adhere to what I call the "Heap Rule." Fortunately, the Heap Rule is pretty straightforward:

- The Heap Rule: Each node's priority must be greater than (or equal to) its parent's priority.

Take a look at the diagram of our example treap. Note that the treap follows the Heap Rule, as the root node contains the smallest priority, and as we descend downward through the tree, the nodes' priorities become increasingly greater. There isn't a single node that has a smaller priority than its parent.

The Heap Rule is derived from the concept of the *heap* data structure, which I covered in Volume 1, Chapter 16. There, I explained that in a min-heap, the value of each node must be greater than its parent's value.

However, with a heap, it was the *value* that needed to follow the Heap Rule. With treaps, on the other hand, it's not the value, but the *priority* that must follow the Heap Rule.

Recall from Volume 1 that a heap can either be a min-heap or a max-heap. The two types of heaps are virtually the same, except that a min-heap makes the arbitrary decision that each node's value must be *greater* than its parent, and a max-heap makes the decision that each node's value must be *smaller* than its parent.

A similar arbitrary decision can be made with treaps. That is, a treap can also be either a "min-treap" or a "max-treap." In this chapter, we'll arbitrarily go with the "min-treap" flavor, simply because much of the treap documentation out there does as well.

The Heap Rule and BST Rule

I noted at the beginning of this chapter that the name "treap" comes from combining the terms "tree" and "heap." This, indeed, is because treaps follow the Heap Rule, much in the way that heaps do. However, I must point out a critical distinction between treaps and heaps.

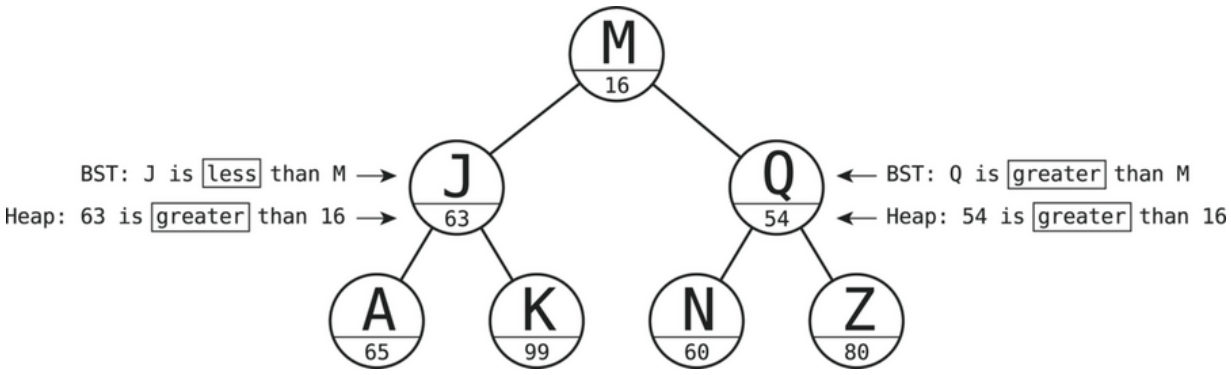
Heaps *only* follow the Heap Rule, and cannot be a BST. Take a min-heap, for example. The heap is only valid if each node has a smaller value than either of its children. However, in a BST, a node is required to have a *greater* value than its left child.

Treaps, though, follow both the Heap Rule *and* the BST Rule. While that might sound impossible at first, here's how this works: treaps follow the BST Rule with regard to the nodes' *values*, but follow the Heap Rule with regard to the nodes' *priorities*.

In other words, treaps must ensure that the *values* of each node act like a BST—namely, that a node's left descendants have smaller values, and a node's right descendants must have greater values. At the same time, treaps

have to be careful that each node's *priority* is greater than the priority of its parent.

Let's take one more look at our example treap. If you look carefully, you'll see that both the BST Rule and Heap Rule are adhered to by all the nodes in the treap:



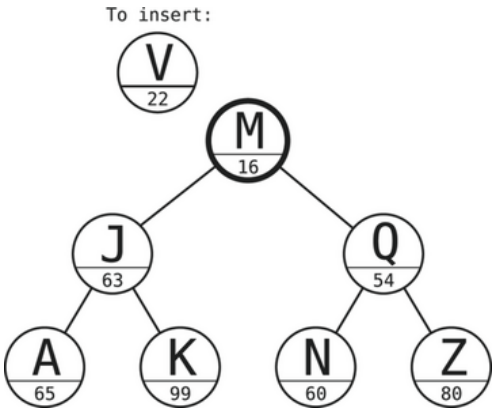
Treap Insertion

We're now ready to see how treap insertions work. Like a red-black tree, treap insertion goes through two phases (as discussed in *Red-Black Tree Insertion*). In the first phase, the new node is inserted in the same way a node is inserted into a regular BST, and in the second phase, the tree is "fixed" so that it doesn't violate any of the rules.

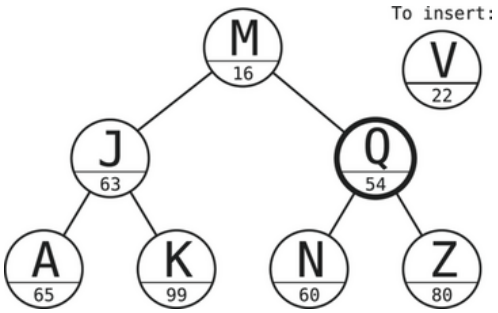
Let's insert the letter V into the treap shown at the end of the previous section. I mentioned earlier that with randomized treaps, a node's priority is randomly generated. So, let's say that the randomly generated priority is 22.

The First Phase

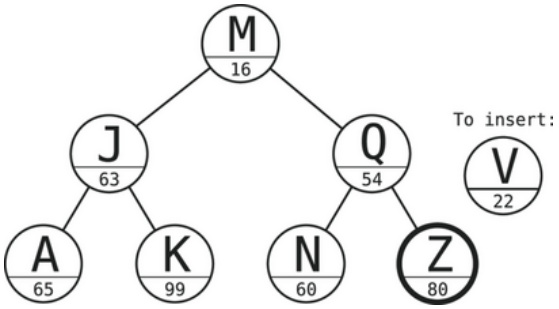
To begin the first phase of insertion, we first compare the V against the root's value of M:



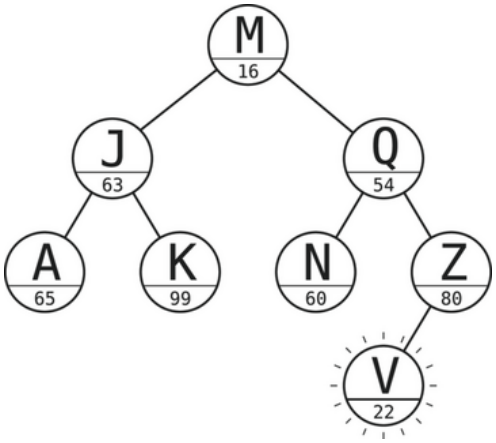
Because the value V is "greater" than the value M, it means that the V will be inserted somewhere among the V's right descendants. So we focus on the M's right child:



We compare the V to the Q. V is greater than Q, so we focus on the Q's right child:



We compare the V to the Z. Because V is *less* than Z, we focus on the Z's left child. However, the Z doesn't *have* a left child, so we insert the V into the tree as the Z's left child:

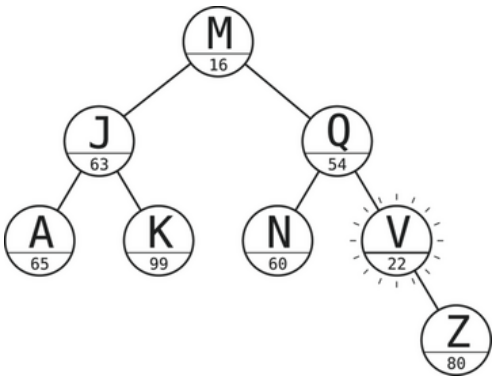


This concludes the first phase of insertion, and you'll notice that it's identical to regular BST insertion. However, you can also see that the Heap Rule has now been violated since the V's priority of 22 is less than its parent's priority of 80. We now need to fix this violation, so the second phase—which I call the "fixing phase"—begins.

The Fixing Phase

At the moment, the V's priority is in conflict with the Z's priority. To fix this, we *rotate* the V and the Z.

Treap rotations are completely identical to red-black tree rotations, and as you saw in the previous chapter, there are two types of rotations: clockwise and counterclockwise. When rotating two nodes (a child and parent), we perform a clockwise rotation when the child node is the *left* child of its parent, and a counterclockwise rotation when the child node is the *right* child of its parent. In this case, the V is the left child of the Z, so we perform a clockwise rotation:



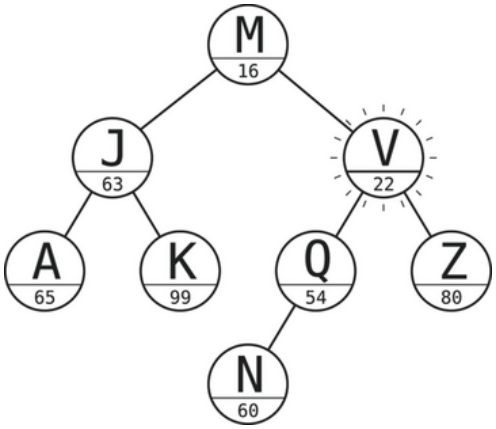
By performing this rotation, we "fixed" the Heap Rule violation between the V and the Z since the V (whose priority is less than the Z) is now above the Z.

At the same time, we can be assured that our fix will not create a violation of the BST Rule since rotations never do. (We covered this in the last chapter, in *Rotations—Part 2*).

It turns out that rotations are a great tool for fixing Heap Rule violations since we can easily "bubble up" a node to its proper place in terms of priority, and not worry that we'll accidentally violate the BST Rule.

Now, while we've fixed the Heap Rule violation between the V and Z, the V still violates the Heap Rule with regard to the Q. That is, the V's priority

of 22 is less than its parent's priority of 54. As such, we now need to perform a rotation between the V and the Q. Because the V is the Q's *right* child, we perform a *counterclockwise* rotation:



The V's priority of 22 is greater than its parent's priority of 16, so the Heap Rule has been restored. Our treap is now fixed, and our insertion is complete!

Self-Balancing Treaps in Action

Because a node's priority is randomly generated, the number of rotations that occur upon treap insertion is determined by randomization. For instance, if our V node from the previous example happened to have been randomly assigned the priority of 100, we wouldn't have performed any rotations at all. This is because a node with a priority of 100 indeed belongs at the bottom of the treap. And if the V received the priority of 1, we would have rotated the V until it became the root of the treap.

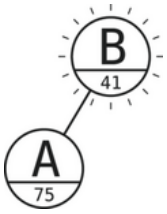
The crazy thing is that this randomization allows treaps to achieve a level of balance that is similar to red-black trees. While red-black trees have a rotation scheme that is complex and follows a precise set of rules, a treap's random rotations achieve a similar result!

Let's look at an example of a treap performing this balancing act. Let's say that we're going to insert values into a treap *in order*. We know that for a regular BST, inserting values in order is a death knell, as the tree becomes a super-long linked list. But let's see what happens with a treap when we insert the values A through G in perfectly ascending order.

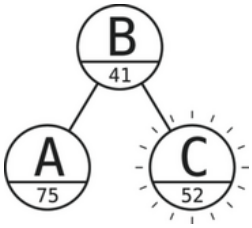
We'll begin by inserting an A. Let's say that this node's randomly generated priority is 75:

Next, we'll insert B. The computer spins its internal die and decides that the B's priority should be 41.

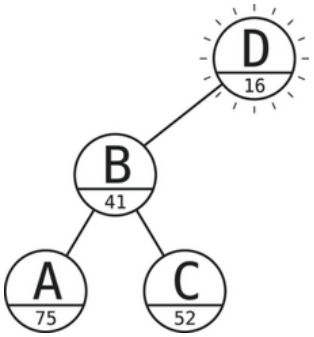
In a regular BST, the B would become the A's right child. However, in a treap, this violates the Heap Rule since the child's priority 41 is less than the parent's priority of 75. And so, we rotate the A and B:



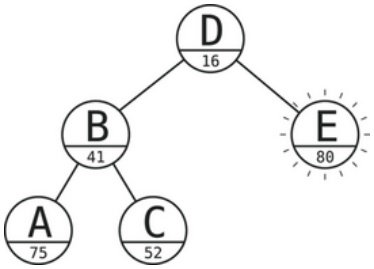
Next up, we have a C, and its randomized priority happens to be 52. As such, we get to make C the B's right child, and no rotations are necessary:



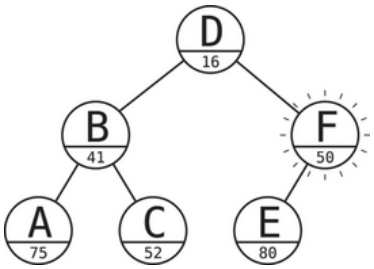
We then insert a D, and the computer decides to assign it a priority of 16. This is the minimum priority of the entire treap so far, so we rotate the D upward until it becomes the treap's root:



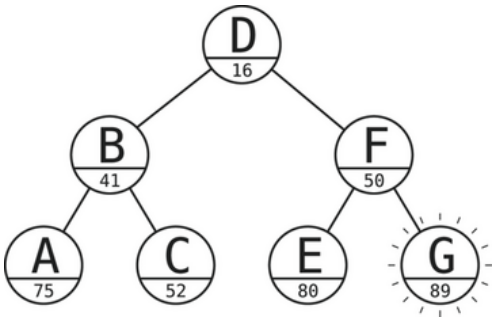
Our next node has the value of E and a priority of 80. No rotations are necessary:



The letter F is up next, and its priority is 50. A single rotation is in order:



Lastly, we insert the G. The computer grants it a priority of 89, so we can simply insert it like this:



Amazingly, although we inserted values in perfect order, the treap rotations arranged them so that our treap is fully balanced.

Of course, this example was completely contrived since I had the liberty to choose the computer's "random" priorities. However, it does turn out that treaps in general have a high probability of being well-balanced.

Let's dig a little further to see why.

The Power of Random Priorities

As I noted earlier, when we insert values in *random* order into a BST, the BST will likely be well-balanced. I'll call this data structure a *randomized BST.*

I'll now demonstrate intuitively why randomized treaps should be as well-balanced as randomized BSTs. The key is that the two data structures share an attribute, which is that *all the values have equal odds of landing at any particular spot within the tree*. Here's what I mean.

Say that we're inserting the values 1 through 100 into a regular BST. When we insert the values in perfectly ascending order, we are dictating that the 1 becomes the root of the tree. This already causes a severe imbalance since the root can now only have right descendants because it's impossible for the 1, the smallest of our values, to have a left child.

But with a randomized BST, each and every value has an equal chance of becoming the tree's root. That is, each value has an equal chance of being picked first, and whichever value is picked first becomes the root.

This alone already helps achieve balance because even though having a 1 as the root is a terrible outcome, there's only a 1 in 100 chance that this will happen. There are greater odds that a more reasonable number will be chosen as the root.

The same goes for randomization at every level of the tree. Regarding the tree's second level, for example, there are some bad numbers that could be chosen, but odds are that this won't happen.

So, the balance of a randomized BST is based on the fact that each and every value has an equal chance of landing at *any* particular level within the tree.

The same goes for randomized treaps: the level where any node lands is completely based on its priority. And because the priorities are random, any value can land at any level.

For example, whichever node has the smallest priority will become the treap's root since otherwise we'd be violating the Heap Rule. And with a randomized treap, each value has an equal shot at being assigned the smallest priority. It might be a bad thing if our smallest value (such as the letter A) also happens to get assigned the smallest priority, but the odds are that this won't happen.

It emerges that both a BST and a randomized treap share the characteristic that all values have the same odds of landing at any particular level within the tree. Because of this, the two data structures have the same level of balance.

The Expected Height of a Treap

Speaking of treap balance, how well-balanced can we expect a treap to be? You discovered in the previous chapter that if a red-black tree has N values, it will have a height of O(log N) if we express the height using Big O notation. Furthermore, it's *guaranteed* that a red-black tree's height will not exceed 2 log N. The Red-Black Rules ensure that a greater height is simply not possible.

Believe it or not, a randomized treap *also* has an expected height of O(log N), making a treap a great competitor to a red-black tree.

However, a treap does not have the same O(log N) *guarantee* that a red-black tree does. It is *possible*, albeit highly unlikely, for a treap to have a much greater height. Imagine that we inserted values in perfect order, and the computer happened to randomly choose priorities that were *also* in perfect order. We'd end up with the dreaded linked list! The good news, though, is that this scenario is *extremely* rare.

I've run many tests building treaps of various sizes (from 50 up to 5,000,000 nodes), and have found that the height of each treap has always come out to be between 2 log N and 3 log N. Again, it's theoretically possible to have a taller treap, but such a likelihood is very small.

It turns out that by using a simple randomized algorithm, treaps perform virtually as well as red-black trees. And not only that, treaps achieve this performance with so much less complexity than the byzantine set of algorithms that power red-black trees.

Yes, red-black trees are never taller than 2 log N, and treaps' heights are usually around 2.5 log N. However, it may be a worthwhile trade-off to have slightly higher trees and thereby gain a considerable amount of code simplicity. More simplicity generally means fewer bugs. And the difference between 2.5 log N and 2 log N may be considered negligible for many applications.

However, if you need a guarantee that your tree doesn't exceed a height of 2 log N, then you may opt for the red-black tree rather than the treap. Again, it's all about trade-offs.

In any case, treaps are a great example of how randomization can achieve simplicity. It's also a perfect example of a data structure that is powered by randomization, or what some call a *randomized data structure*.

Code Implementation: Treap Insertion

Let's look at the Python code for treaps. To start, here's the code for a treap node:

```
import random


class Node:
    def __init__(self, value, priority=None):
        self.value = value
```

```
        self.priority = priority or random.random()
        self.left_child = None
        self.right_child = None
        self.parent = None
```

Like other tree nodes we've worked with, this `Node` has `value`, `left_child`, `right_child`, and `parent` attributes. However, what makes treap nodes unique is that they also have a `priority` attribute.

In a randomized treap, this `priority` will simply be a random number. In our implementation, we use the `random()` method to generate this number, which will be a random float between 0 and 1. That is, it'll be something like `0.2677900713452904` or `0.9337348703962393`.

Mainly for testing purposes, I built in the ability for the creator of a node to assign a priority to that node. A random priority will only be generated if no other priority has been explicitly assigned. This also allows our treap to be used in its classical variant, where the priorities are not random. And so, our code can be used to serve either as a randomized treap or a classic treap.

Now that we have our treap `Node` in place, here is the first section of code for our actual `Treap` class:

```
import treap_node


class Treap:
    def __init__(self, root=None):
        self.root = root

    def rotate_counterclockwise(self, a, b):
        a.right_child = b.left_child

        if b.left_child:
            a.right_child.parent = a

        b.parent = a.parent
        if not b.parent:
```

```
            self.root = b
        elif b.parent.left_child == a:
            b.parent.left_child = b
        else:  # Node B is a right child
            b.parent.right_child = b

        b.left_child = a
        a.parent = b

    def rotate_clockwise(self, b, a):
        b.left_child = a.right_child

        if a.right_child:
            b.left_child.parent = b

        a.parent = b.parent
        if not a.parent:
            self.root = a
        elif a.parent.right_child == b:
            a.parent.right_child = a
        else:  # Node A is a left child
            a.parent.left_child = a

        a.right_child = b
        b.parent = a

    def is_a_left_child(self, node):
        return node == node.parent.left_child

    def is_a_right_child(self, node):
        return node == node.parent.right_child

    def insert(self, value, priority=None):
        new_node = treap_node.Node(value, priority)

        if not self.root:
            self.root = new_node
            return

        current_node = self.root

        while current_node:
            if value < current_node.value:
```

```
                if not current_node.left_child:
                    current_node.left_child = new_node
                    new_node.parent = current_node

                    current_node = current_node.left_child

                elif value > current_node.value:
                    if not current_node.right_child:
                        current_node.right_child = new_node
                        new_node.parent = current_node

                    current_node = current_node.right_child

                else:  # value is already inside tree
                    break

            self.insert_fix(new_node)

        def insert_fix(self, node):
            while node.parent and node.priority < node.parent.priority:
                if self.is_a_left_child(node):
                    self.rotate_clockwise(node.parent, node)
                else:
                    self.rotate_counterclockwise(node.parent, node)
```

This is a lot of code, but much of it is code we've encountered before. Let's walk through all of the code in order.

The `Treap`'s constructor creates the `self.root` variable, which is used to keep track of the treap's root at all times.

The bulk of the remaining code consists of several methods that I copied from our red-black tree implementation in the previous chapter in *Code Implementation: Nodes and Rotations*. These include `rotate_counterclockwise`, `rotate_clockwise`, `insert`, and smaller helper methods `is_a_left_child` and `is_a_right_child`.

The main difference compared to our red-black tree code is the implementation of the `insert_fix` method. The `insert_fix` algorithm here is:

```
while node.parent and node.priority < node.parent.priority:
     if self.is_a_left_child(node):
          self.rotate_clockwise(node.parent, node)
     else:
          self.rotate_counterclockwise(node.parent, node)
```

We run a loop as long as the inserted node's priority is less than its parent's priority. In the loop, we continue to rotate the inserted node with its parent. The rotation will be clockwise if the inserted node is a left child and counterclockwise if the inserted node is a right child. Once the inserted node is situated so that its priority is greater than (or equal to) its parent's priority, we're done.

It's worth taking a peek back at the red-black tree's `insert_fix` code in *Code Implementation: Red-Black Tree Insertion (Fixing Phase)* to appreciate how much simpler the treap algorithm is. It's pretty incredible what randomization can do.

Treap Deletion

Deleting a node from a treap is also much simpler than it is with a red-black tree. The trick is that we can use rotations to move the node we're deleting *downward* through the treap until it becomes a leaf node (that is, it has no children). Once this node becomes a leaf, we simply detach it from the treap.

Here's the specific algorithm:

1. First, we search the tree for the node with the value that we intend to delete. We'll call this node the `node_to_delete`.

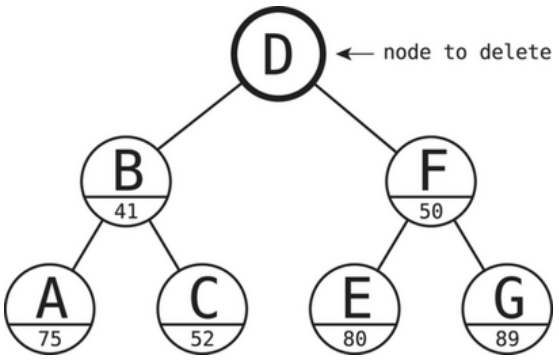
2. If we find this node, we run a loop that lasts as long as the `node_to_delete` has children.

3. Within this loop, we rotate the `node_to_delete` with its *child*. If the `node_to_delete` happens to have two children, we rotate it with the child that has the *smaller* priority. (I'll explain the reason for this shortly.) We continue to rotate the `node_to_delete` downward until it has no children, after which our loop ends.

4. We remove the `node_to_delete` from the treap. That is, if the `node_to_delete` is a left child, we declare that its parent's left child is henceforth `None`, and if the `node_to_delete` is a right child, its parent's right child will now be `None`. By doing this, we can no longer access the `node_to_delete` from the treap.

Treap Deletion in Action

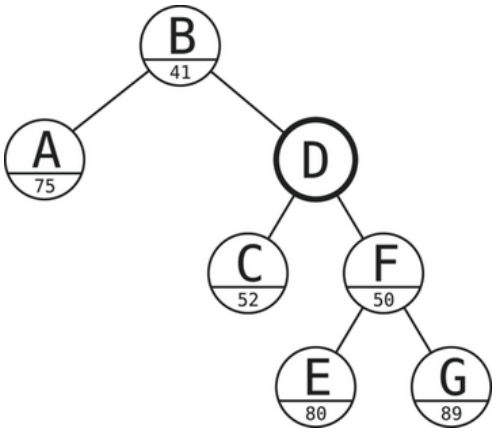
Let's get a visual of treap deletion so that the algorithm is abundantly clear. Say we want to delete the D node from the following treap:



Note that in this diagram, the priority of the D node is obscured. This was done because the D's priority is completely irrelevant in the deletion algorithm. That is, you don't ever need to look at the D's priority to perform deletion.

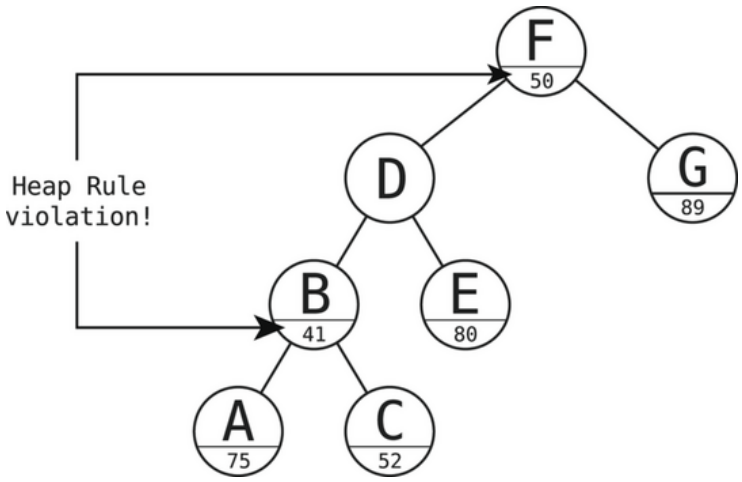
To delete the node from the treap, first, we conduct a search to find the node. The search algorithm is the same algorithm for searching for a node within a classic BST. In this example, we find the D immediately since it happens to be the root node.

We then begin a loop that rotates the D downward through the treap. The D currently has two children. The B has a priority of 41, while the F has a priority of 50. As stated earlier, we choose to rotate the `node_to_delete` with the child with the *smaller* priority. This would be the B since 41 is less than 50. So, we rotate the D and the B in a clockwise fashion:



Let me pause here to explain why we chose to rotate the `node_to_delete` with the child with the *smaller* priority.

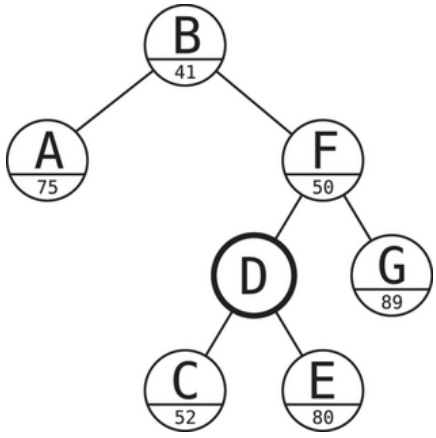
If we do the alternative, and rotate the `node_to_delete` with the child with the *greater* priority, we'd end up moving the F upward, like so:



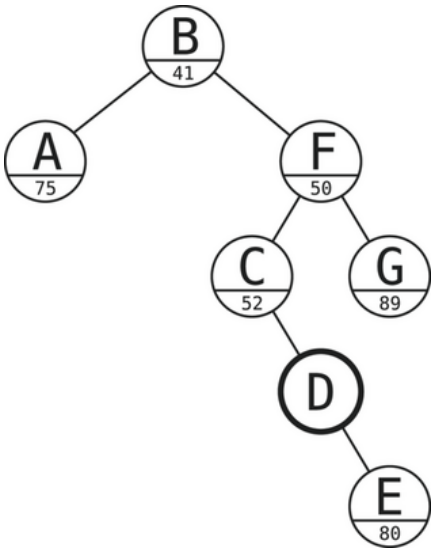
This would put us in a position where the F is an ancestor of B. However, because F has a greater priority than B, this violates the Heap Rule!

When we rotate a node downward, we end up turning one of its children into the ancestor of the other child. By rotating the node with the child with the smaller priority, we ensure that the child with the smaller priority becomes the ancestor of the child with the greater priority.

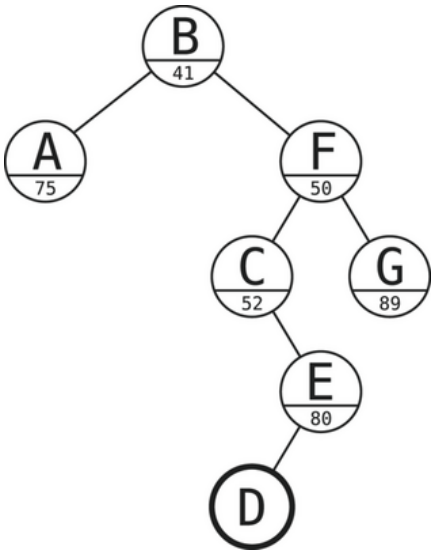
Let's go back to where we were when we did things correctly. Currently, the D has two children, C and F. Between the two children, F has the smaller priority, so we rotate the D with the F in good ol' counterclockwise fashion:



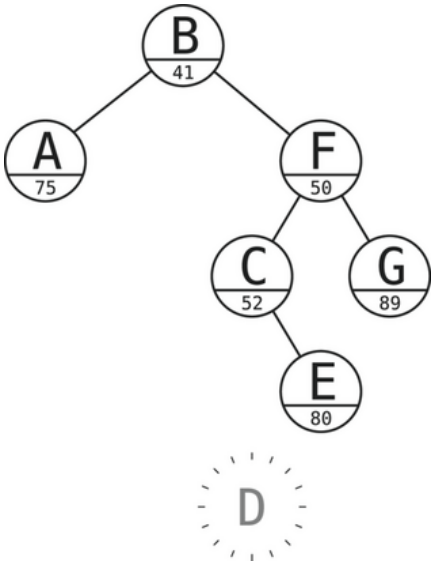
The D's two children are now C and E. The C's priority is smaller than the E's, so we rotate D and C clockwise, as shown in the treap.



The D now only has one child, E, so we rotate D and E:



The D is now a leaf node, so we can now safely remove it from the treap:



Poof! It's gone.

Treap Deletion vs. BST Deletion

In Volume 1, Chapter 15, I covered how BST deletion works. I won't rehash all the details here, but the algorithm is arguably more complex than the algorithm for treap deletion. In particular, the BST deletion algorithm performs different actions in different scenarios, such as when the `node_to_delete` has no children, one child, or two children. It also deals with moving around a "successor node" and managing the successor node's child if it has one.

Treap deletion, on the other hand, can pretty much be summed up in one sentence: *Continuously rotate the `node_to_delete` downward with the child with the lesser priority until the `node_to_delete` becomes a leaf node, and then remove it from the treap.*

Now, here's the interesting thing. The treap deletion algorithm would work as well on a classic BST, too. Although BSTs don't have priorities, we could simply rotate the `node_to_delete` downward through the tree, choosing any child at random, until the `node_to_delete` becomes a leaf node, after which we can remove it from the tree. Again, rotations never cause a

violation of the BST Rule, so there's no reason we can't perform this deletion algorithm on a BST.

It seems that the only reason this algorithm is not typically used on classic BSTs is that it involves rotations, which are not themselves the simplest of all algorithms. Essentially, then, it's a complexity trade-off. The BST deletion algorithm has complexity in dealing with multiple scenarios but avoids having to implement rotations.

Treaps, on the other hand, already have to implement rotations to power insertions. Once the ability to perform rotations is in place, we may as well use them to implement a simple deletion algorithm.

On a similar note, we could also technically use the classic BST deletion algorithm on treaps, but this would, in many cases, trigger a Heap Rule violation since we often plug a "successor node" into a higher spot within the tree. We'd then have to fix the treap by rotating the successor node down through the treap. But this is all considerably more complicated than simply rotating the `node_to_delete` down through the treap in the first place, so why go through all that trouble?

Code Implementation: Treap Deletion

Here, we implement a `delete` method, which in turn depends on a `search` method:

```
    def search(self, value):
        if not self.root:
            return None

        current_node = self.root

        while current_node:
            if value < current_node.value:
                current_node = current_node.left_child
            elif value > current_node.value:
                current_node = current_node.right_child
```

```
            else:  # value found!
                return current_node

    def delete(self, value):
        node_to_delete = self.search(value)

        if not node_to_delete:
            return False

        if node_to_delete == self.root \
          and not node_to_delete.left_child \
          and not node_to_delete.right_child:
            self.root = None
            return node_to_delete

        while (node_to_delete.left_child or node_to_delete.right_child):
            if (not node_to_delete.right_child) \
              or (node_to_delete.left_child.priority <
                  node_to_delete.right_child.priority):
                self.rotate_clockwise(node_to_delete, node_to_delete.left_child)
            else:
                # this else clause occurs if either there is only a right child
                # or the right child has smaller priority than left child
                self.rotate_counterclockwise(node_to_delete,
                                             node_to_delete.right_child)

        if self.is_a_left_child(node_to_delete):
            node_to_delete.parent.left_child = None
        else:
            node_to_delete.parent.right_child = None

        return node_to_delete
```

The `search` method is identical to classic BST search, so let's focus on the `delete` method.

We begin by searching for a node with the value we're trying to delete. If we find that node, it's assigned to the variable `node_to_delete`. If it's not found, we terminate the method early by returning `False`:

```
        node_to_delete = self.search(value)
```

```
if not node_to_delete:
     return False
```

Next, we handle the unique case where the `node_to_delete` is the only node inside the treap, in which case we simply remove it from the treap by resetting the `root` to `None`:

```
if node_to_delete == self.root \
  and not node_to_delete.left_child \
  and not node_to_delete.right_child:
    self.root = None
    return node_to_delete
```

Next, we begin our loop that runs as long as the `node_to_delete` has at least one child:

```
while (node_to_delete.left_child or node_to_delete.right_child):
```

We then perform a clockwise rotation with the `node_to_delete` and its left child if either there's no right child or if the left child has a smaller priority than the right child:

```
if (not node_to_delete.right_child) \
  or (node_to_delete.left_child.priority <
node_to_delete.right_child.priority):
    self.rotate_clockwise(node_to_delete, node_to_delete.left_child)
```

On the other hand, if only a right child exists or if the right child's priority is smaller than the left child, we rotate the `node_to_delete` with its right child:

```
else:
     self.rotate_counterclockwise(node_to_delete,
node_to_delete.right_child)
```

Finally, we cut off the `node_to_delete` from the treap:

```
if self.is_a_left_child(node_to_delete):
     node_to_delete.parent.left_child = None
else:
```

```
            node_to_delete.parent.right_child = None
```

We finally conclude the method by returning the `node_to_delete` to confirm that the deletion was successful.

And that's it!

Wrapping Up

Randomized treaps are a great example of a randomized data structure and, more importantly, a great example of how effective randomization is in simplifying algorithms. Red-black trees are notorious for making people's heads spin, and yet treaps achieve similar results with insertion and deletion algorithms that require considerably fewer steps and conditions. We looked at some practical applications of red-black trees in the previous chapter, and randomized treaps can be used wherever red-black trees can.

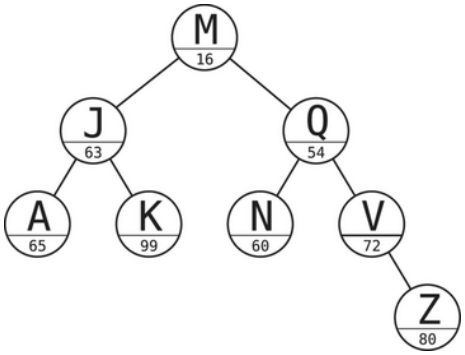
In the next chapter, we're going to take a slight detour from the theme of randomization. Because we're already dealing with self-balancing trees, I want to introduce you to another such tree known as the B-tree. Besides being extremely prevalent in all sorts of applications, this topic will open up an entirely new class of problems and algorithms—and disrupt the way we think about time complexity in general.

I'll see you there.

Exercises

The following exercises provide you with the opportunity to practice with treaps. The solutions to these exercises are found in the section *Chapter 6*.

1. Look at this shiny randomized treap:



   What will the treap look like after inserting a node with the value `"L"` and the priority 100?

2. What will the previous treap look like if the `"L"` node we're inserting has a priority of 5 (rather than the 100 from the previous exercise)?

3. What will the previous treap (had you not done the insertion of the previous exercises) look like if we delete the `"Q"` node?

Chapter 7

To B-Tree or Not to B-Tree: External-Memory Algorithms

Throughout our journey so far, we've been operating under the assumption that our algorithms are dealing with data that is contained completely within the computer's main memory (otherwise known as RAM) or caches. In other words, our computer has all the data loaded in memory and ready to go, and the computer then performs an algorithm on that data.

However, this isn't always the case. Let's say we want to calculate the sum of a list of integers that is so long that it takes up 50GB of space. If we're working with a computer that has only 8GB of main memory, we immediately encounter a problem. How is our computer supposed to process such a list if the list can't even fit inside the computer's active memory? It's not even possible to declare the statement:

```
array = [6, 2, 0, 1, 8... super long list that is 50GB long]
```

While our computer probably wouldn't explode if we attempted this, the computer *would* reject the statement, whining that it can't hold so many numbers.

In this chapter, you'll learn how to properly analyze the efficiency issues surrounding "big data" problems such as this one and write effective

external-memory algorithms to process such data quickly and with minimal space consumption.

Another important problem we'll deal with in this chapter is how to manage tree data structures when dealing with massive amounts of data. As you've seen throughout our discussions of trees, a tree such as a BST (be it a classic one, a red-black tree, or a treap) is important for being able to search for data quickly while also keeping the data sorted. But what if our data is so large that our tree can't fit inside main memory? To deal with this not-uncommon conundrum, you'll learn about the important and ubiquitous B-tree, which is a tree specialized for storing *lots* of information.

In this as well as the next chapter, we'll embark on a "side quest" in which we explore the fascinating, vast, and important world of external-memory algorithms. This chapter is indeed connected to the previous ones, as here we'll feature another self-balancing tree as we did in the previous two chapters. However, we won't return to our main theme of randomization again until Chapter 9. If you so choose, you could skip ahead at this point to Chapter 9 and return to Chapters 7 and 8 at some later time.

External Memory

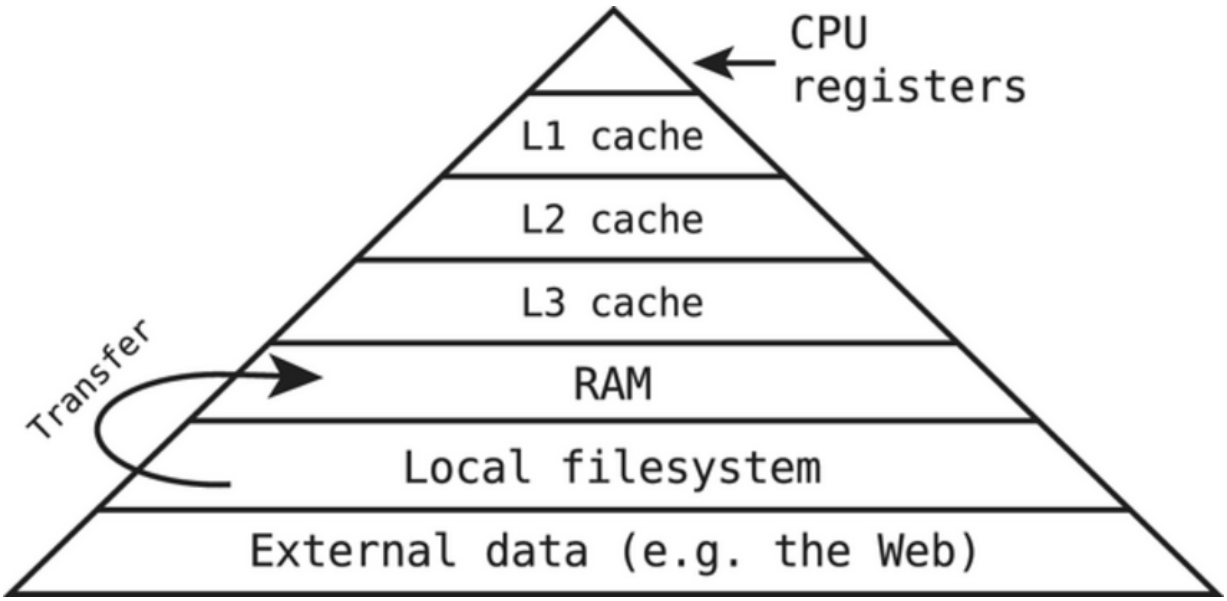
Let's get back to our 50GB list of integers. To store such a list, we need to store the integers somewhere *outside* main memory since the list certainly won't fit *inside* main memory. Typically, we'd do this in the filesystem, assuming that our filesystem *is* larger than 50GB.

More specifically, we'd create a file that stores all the integers. Given that such a file would itself be massive, we might also decide to break up the data into several smaller files.

In any case, though, this approach alone doesn't fully solve our problem. This is because when we run code, the computer works directly with data in *main memory*, and doesn't deal directly with data from the filesystem. We touched on this earlier in *The Memory Hierarchy*, but let me review these details and elaborate on them a bit further since these concepts are the foundation of this chapter.

You can think about a computer's filesystem as a storage center, similar to those walk-in storage centers where you keep physical stuff you don't use on a regular basis. If I store a sofa in such a storage center, I will never go to the storage center to sit on that sofa. Similarly, when code performs an algorithm, the software doesn't act directly on data in the filesystem "storage center." Instead, the code copies data from the filesystem into main memory and only *then* does our code do something with that data.

Using the memory hierarchy diagram you first saw here, we're essentially referring to the transfer of data as shown in the figure.



In truth, some of the data may also be copied into the various cache levels. When I refer to either "main memory," "RAM," or even simply "memory" in this chapter, I mean to include the cache levels as well.

So, we return to our million-dollar question: how can our code work with data if the data is too large to fit inside RAM?

Memory Blocks

Fortunately, computers have a built-in technique for handling this kind of thing. When a computer loads data from the filesystem, it does so one *block* at a time. The term *block* refers simply to a section of data. Taking our example of a 50GB list of integers, the computer might view it as a collection of fifty 1GB blocks.

The exact size of a block depends largely on your particular computer's hardware and operating system. Because of this, the computer, by default, generally uses the same block size across all pieces of software.

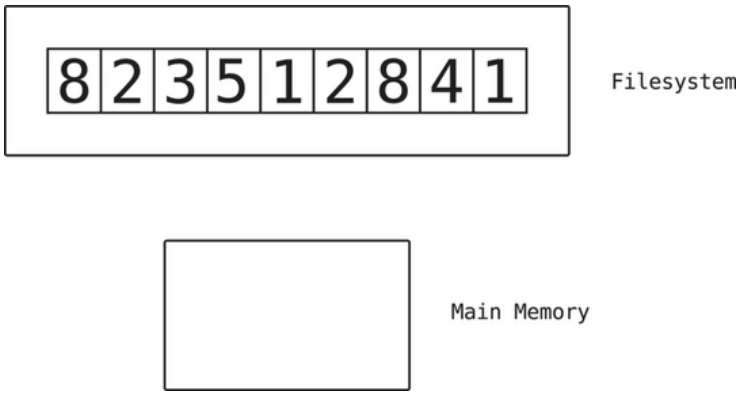
Now, let's say that we're working with a computer that has 8GB of RAM and a block size of 1GB. (The filesystem can be infinitely large for the purposes of our discussion.) Here's the gist of how the computer may compute the sum of integers contained in a 50GB file:

Because our computer has 8GB RAM, the computer will start out by loading the first 8 blocks of data from our 50GB file, with each block containing 1GB of data. This will fill the capacity of our RAM since our RAM is 8GB, and the 8 blocks contain a combined total of 8GB of data. Our algorithm will then compute the sum of those integers and keep that sum in a variable. This variable, essentially, tracks the "total sum so far."

Next, the computer will eject those first 8 blocks from memory, freeing up our RAM, and then load the next 8 blocks from the file. We'll compute the sum of those freshly loaded integers and add the result to the "total sum so far." We repeat this process until we've processed all 50GB worth of integers. Once we've completed this process, our "total sum so far" variable will actually contain the total sum of the entire 50GB list of integers. And so, we've succeeded at our goal of summing up the entire massive list.

Let's look at a high-level visual of the algorithm I've just described. However, to keep the diagrams small enough to fit on the page, I'm going to change the size of our main memory and blocks. Imagine that our computer is so tiny that its RAM can store only 3 integers. At the same time, let's also say that the block size, too, is 3 integers.

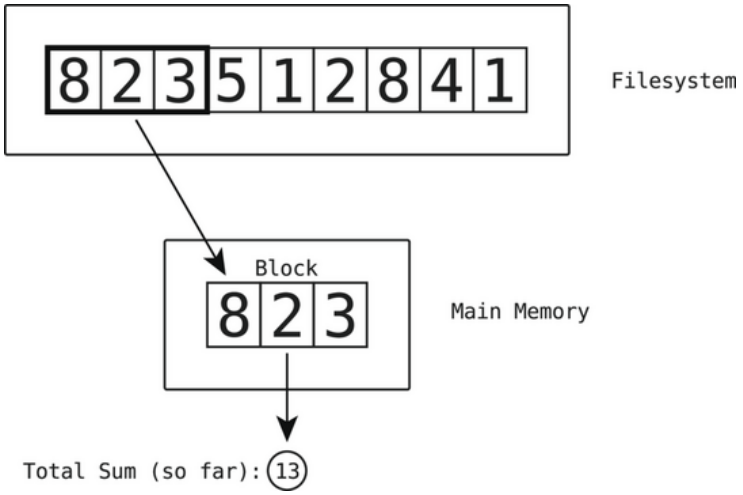
Here's a depiction of our computer's main memory and a file in the filesystem containing integers that we want to sum up:



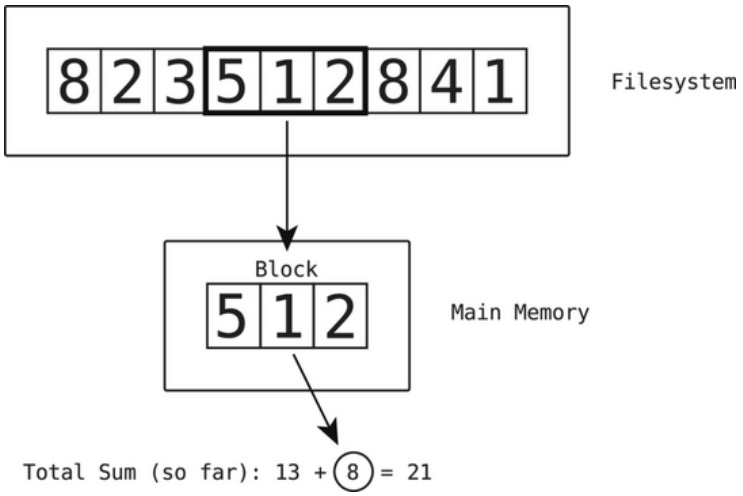
Currently, the main memory is empty.

Our computer can't calculate the sum in the classic way, since it can only work with 3 integers in main memory at once. Again, this is because our example computer's RAM has a maximum capacity of 3 integers.

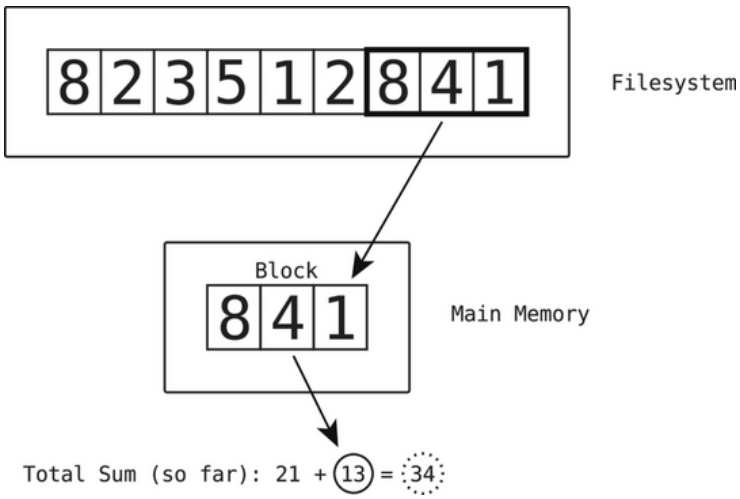
So, we first have to load a block of 3 integers from the file and compute the sum of those 3 integers:



We then load the second block from the file, sum *those* integers, and add that to our total sum so far, as shown in the first diagram.



Finally, we do the same with the last block as shown in the second diagram.



And the algorithm is complete.

We can refer to an algorithm like this as an *external-memory algorithm* since it's an algorithm that processes data from an external source, such as the filesystem.

What's sweet is that for many of Python's built-in tools for loading data from files, the computer manages all of this block maneuvering *automatically*, and we don't have to write explicit code to move blocks of data.

It might seem that we can continue to write our code the same way we've always done and remain completely oblivious to the way our computer transfers data blocks from the filesystem to main memory. However, you'll soon see that for many applications, this couldn't be further from the truth.

Slow I/Os

It's a bit of a mouthful to keep saying the phrase "transfer data from the filesystem to main memory." So, let's use a shorthand term. An *I/O*, which stands for "input/output," is an operation that transfers computer data. It can refer to various types of data transfers in different contexts, but in this chapter, I'll use it to refer to transferring data from the filesystem to main memory.

Now, while I'd love for you to remember *everything* I ever write (including the jokes), if there's only one thing you can remember from this chapter, it should be this: *I/Os are extremely slow!*

A major theme of this book has been that we measure an algorithm's time complexity in terms of the number of "steps" the algorithm takes to complete a task. However, if our algorithm involves I/Os, our perspective needs to change. This is because an I/O can be *thousands* of times slower than a typical "step" that takes place in main memory.

This has major ramifications. If we have an algorithm that involves 49 main-memory steps plus one I/O step, we *might* have thought to say that this algorithm takes 50 steps. But this would be a poor reflection of the algorithm's true speed. For if the I/O has the equivalent speed of *1,000* main-memory steps, our algorithm is much slower than 50 steps. It's more like 1,049 steps! There are the 49 main-memory steps, but there's also the single I/O that is as slow as 1,000 main-memory steps.

And so, when we have an algorithm that loads data from the computer's filesystem, our long-standing Big O approach to measure time complexity is no longer as useful as it once was.

However, fear not, for we'll set everything right again. We need to modify the way we *use* Big O notation to accommodate external-memory algorithms.

Count I/Os, Not Steps

Because of the slowness of I/Os, when it comes to algorithms that involve I/Os, computer scientists like to measure time complexity differently than they do with "regular" non-I/O algorithms. Specifically, instead of counting how many *steps* the algorithm takes, we count the number of *I/Os* the algorithm performs.

This line of thinking is driven by how much slower I/Os are when compared with main-memory steps. If an algorithm has 5 I/Os plus 300 main-memory steps, the 5 I/Os are so much slower than the 300 main-memory steps. Therefore, *we simply ignore the main-memory steps altogether*.

This is a bit similar to what I discussed back in Volume 1, Chapter 6, namely, that when we have an algorithm with $N^4+N^3+N^2+N$ steps, we only keep the highest order and simply say that the algorithm is $O(N^4)$. I explained that this is because when compared with N^4, all the lower orders of N are insignificant.

The same basic idea holds true here as well. When our algorithm includes both I/Os plus main-memory steps together, we only count how many I/Os there are since the main-memory steps are insignificant by comparison.

Big O of I/O

In Volume 1, Chapter 3, I explained that Big O notation is valuable because it effectively describes the time complexity of any algorithm, irrespective of how much data the algorithm is processing. Instead of saying that a particular algorithm takes 10 steps for 10 data elements and 1,000 steps for 1,000 data elements, we simply say that the algorithm has a speed of O(N), with N signifying how much data we're dealing with.

But to adapt Big O Notation to external-memory algorithms, we need to switch what it is that Big O measures. Specifically, we will adapt Big O Notation so it tells us *how many I/Os the algorithm performs when there are N elements*.

Let me break this down.

As mentioned, for "regular" algorithms, the term O(N) says that for N data elements, the algorithm takes N steps. But for external-memory algorithms, we'd want O(N) to mean that for N data elements, the algorithm performs N *I/Os*.

But here's the thing. Let's return to our earlier algorithm in which we compute the sum of all the integers in a list. If all the integers were in RAM, the algorithm would indeed be O(N) since for the N integers, the algorithm takes N steps.

But is this true for the external-memory version of summing integers? Is it accurate to say that for N integers, the algorithm performs N I/Os?

Indeed, this is not the case. We saw in our visual example that when there are 9 integers, our algorithm performed only *3* I/Os. So how *do* we use Big O to tell us how many I/Os the algorithm performs when there are N elements?

If you can resist the temptation to read further immediately, I recommend you first take a moment to ponder this and try to come up with your own answer.

Okay, let's analyze this thing.

The key to everything lies in the computer's *block size*. I mentioned earlier that each computer, based on its own particular hardware specs, has a particular block size. If there are 10 pieces of data, and the block size is 2, we know that it will take 5 I/Os to process all the data. In other words,

because each I/O transfers a single block from the filesystem to main memory, it'll take 5 I/Os to transfer all the blocks of size 2 until all 10 pieces of data are transferred.

If we want to express this using math, we'd say:

```
10 pieces of data / 2 block size = 5 I/Os
```

To express this same idea in terms of *N* pieces of data, we'd say that:

```
N / block size = total number of I/Os
```

To make this notation a bit more concise, computer scientists like to use the variable "B" to refer to the block size. So, to express the previous formula using this shiny new variable, we'd say:

```
N / B = total number of I/Os
```

There's potential to get confused regarding B, so let me make this abundantly clear: B stands for block *size*, and *not* the *number* of blocks. It's easy to mix that up, so go ahead and repeat this factoid 10 times before moving on.

With this, we've now unlocked the ability to express the number of I/Os relative to N. That is, for N data elements, an algorithm will perform N/B I/Os.

We can now answer our original question of how we can use Big O to describe the time complexity of summing all the integers in a file. We'd say that it takes:

O(N/B) I/Os.

This is the Big O way to express that for N data elements, the algorithm will execute N/B I/Os.

Finding Duplicates

Let's look at another instance where we can apply Big O notation to I/Os.

Say that we have an array of strings, and we want to see if we can find any duplicate strings. Throughout Volume 1 (and especially in Chapter 19), I laid out both fast and slow ways of doing this.

The slowest approach is to apply brute force: for each string, check all the other strings to see if we get a match. In a "regular" case, where our array of strings is small enough to fit inside main memory, the speed of this approach would be described as $O(N^2)$. This is because for each of the N strings, we have to check N strings.

But now imagine that our list of strings is so large that it cannot fit in RAM and can only be stored as a humongous file in the filesystem. How would we describe the speed of our brute-force algorithm now? To figure this out, the first thing we need to do is determine what I/Os the algorithm would need to perform. After doing that, we can then count how many I/Os the algorithm performs in total.

We know that the computer will virtually divide the data into blocks and perform an I/O each time to load a block into main memory. To access even the first string in our file, the computer will need to transfer the first block into memory.

Let's say that the first string in our file is `"apple"`. This means that we want to search the rest of our list to see if there's another instance of `"apple"`. Now, because we happen to have the first block in memory, we may as well search that block for another `"apple"`. If there is no such matching string in the first block, the computer must now go to the filesystem again and perform a second I/O to load the next block into memory.

Eventually, the computer will load the entire file one block at a time, assuming it doesn't find a duplicate before getting to the end of the file. Because there are N strings in the file, we'd say that the computer performs

N/B I/Os to perform a search for the duplicate `"apple"`. Now, keep in mind that we've only so far dealt with the search for `"apple"`! In a worst-case scenario, where the list doesn't contain any duplicates, we need to repeat the same approach for *each and every string* in our file.

And so, the algorithm must ultimately perform N/B I/Os for each of the N strings. This is N/B * N, which is equivalent to N^2/B. As such, Big O notation would describe this as $O(N^2/B)$.

The main takeaway from this section is that we've discovered the preferred approach for measuring the time complexity of external-memory algorithms. In a nutshell, it all boils down to counting I/Os rather than counting main-memory steps.

It's certainly great to be armed with this knowledge, but you haven't yet seen how it might dictate the way you should write our code. But that's all about to change now. In the next section, you'll see that some external-memory algorithms can be written in multiple ways and that we'll need Big O to help us determine which approach is fastest.

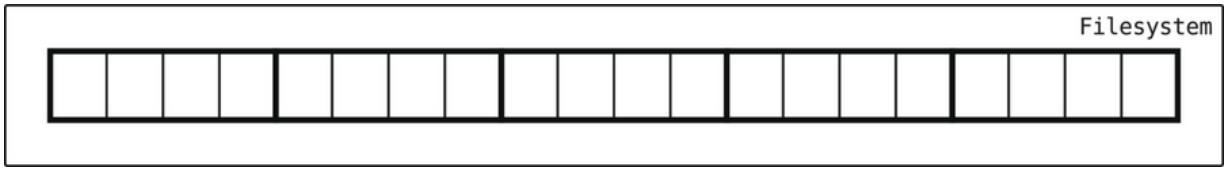
External Binary Search

Ah, binary search—one of the classic algorithms of computer science lore. (I covered it way back in Volume 1, Chapter 2.) It's a super-fast algorithm, clocking in at a blazing O(log N).

The typical binary search deals with data that is small enough to fit inside main memory. But what would binary search look like when applied to *external* memory? That is, say we had an ordered list stored in the filesystem, but the list is too large for RAM. How might we perform binary search?

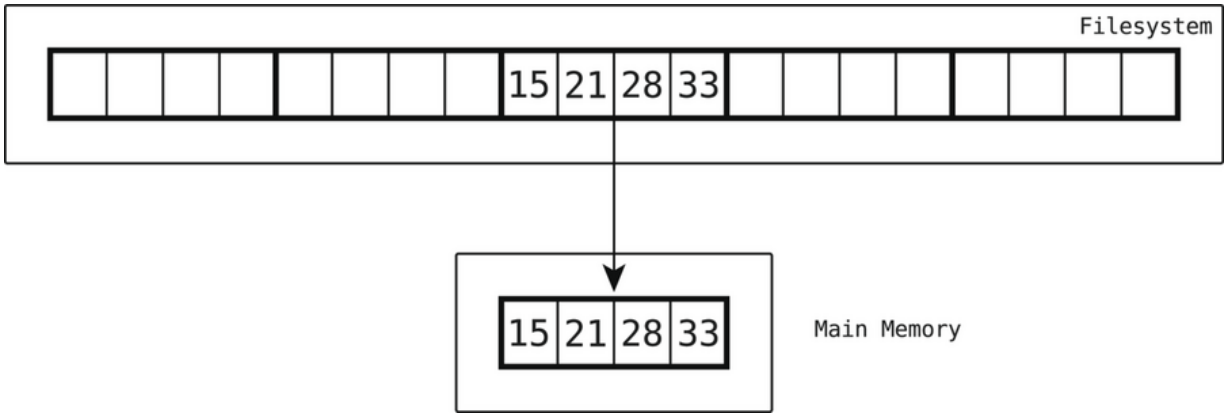
Perhaps the most straightforward approach would be to perform binary search on *blocks of data*. Let's visualize this.

Say that we have an ordered file that contains 20 values, and our computer's block size is 4. This means that we have 5 distinct blocks, each containing 4 values:



In the previous diagram, I am not yet revealing the identity of these integers. I'll reveal them when we load them into main memory.

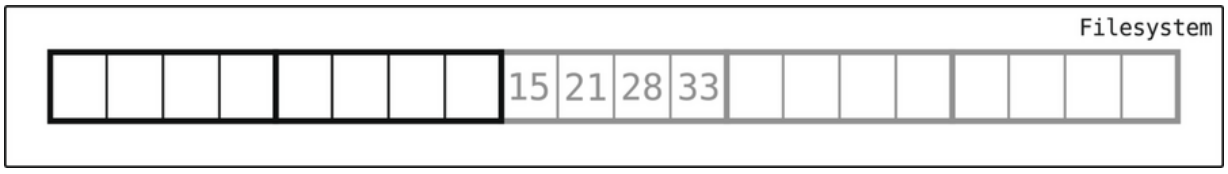
Now, suppose we want to find the integer 11 within our data. With binary search, we always start by inspecting the value that's at the *center* of our list. To do this, we'll have to load the center *block* into main memory:



Once we have this block in main memory, we can now search the current block. We can do this, naturally, with binary search on the elements of the current block.

(A nice little trick, though, is to initially inspect the first and last values of the block. For example, if we're looking for 11, and the first and last values of the block are 15 and 33, respectively, we know right off the bat that the 11 could only lie in a block that is earlier than the current block. However, this trick may only serve to reduce the number of *main memory* steps, and doesn't reduce the number of I/Os we have to perform. So let's move on.)

In any case, the 11 we're seeking is less than any of the elements in the current block. As such, we know that the value must live inside a block earlier in the list, and we can eliminate all other blocks:



We repeat this process by choosing the center block of what remains. In this case, where we have an even number of remaining blocks, we can just arbitrarily choose either of them.

With this process, we end up performing *log N/B I/Os*. That is, we start with N/B blocks, and with each I/O we perform, we reduce the number of remaining blocks by half. So, since in-memory binary search takes log N

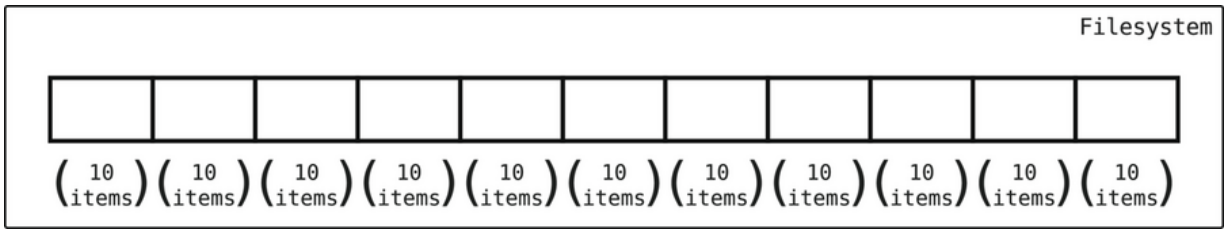
steps, external-memory binary search takes log N/B I/Os. In Big O notation, we'd call this O(log N/B).

In the end, external binary search isn't much different than good ol' regular binary search. It's practically the same algorithm, and has the same logarithmic kind of performance. However, we're going to look at how we can optimize this external memory algorithm and make it *much* faster. I'm going to start with a theoretical approach and then move on to practical techniques we can take to the bank.

Optimizing External-Memory Algorithms

I've demonstrated how binary search works in the context of external memory. Now I'm going to present another example of this, but this time with a larger data set, as shown in the following illustration.

Say that we have a file containing a list of 110 elements, and that our computer's block size is 10. This means we have 11 blocks of data in total. Each rectangle in the diagram that follows represents 10 items of the list; I simply can't fit 110 values into this image:



Now, here's a pop quiz for you (since I know you like them): what is the greatest number of I/Os we'd have to perform by executing binary search on this data set?

The answer is that we'd have to incur at most 4 I/Os. There are 11 blocks, and $\log_2 11$ is about 4.

This isn't bad, but we can do better.

Let's look at an optimization technique that will enable us to search our example file using no more than *2* I/Os. As you'll see, the first iteration of this approach is only theoretical, but you'll see later how it can be used practically. This approach, as well as everything I'll discuss for the remainder of this chapter, will be based on what I call the *key* to optimizing external-memory algorithms.

The Key to Optimizing External-Memory Algorithms

Without further ado, the key to speeding up external-memory algorithms is to *pack as much useful information as we can inside each block*. Let's take a look at what this means, first, in the context of binary search.

When we execute binary search on external memory and perform an I/O to transfer a block into main memory, how much of that block's information is useful to us?

Well, it depends.

If the value we're searching for is inside that block, we only care about that value. All the other data is simply the haystack that hides our needle.

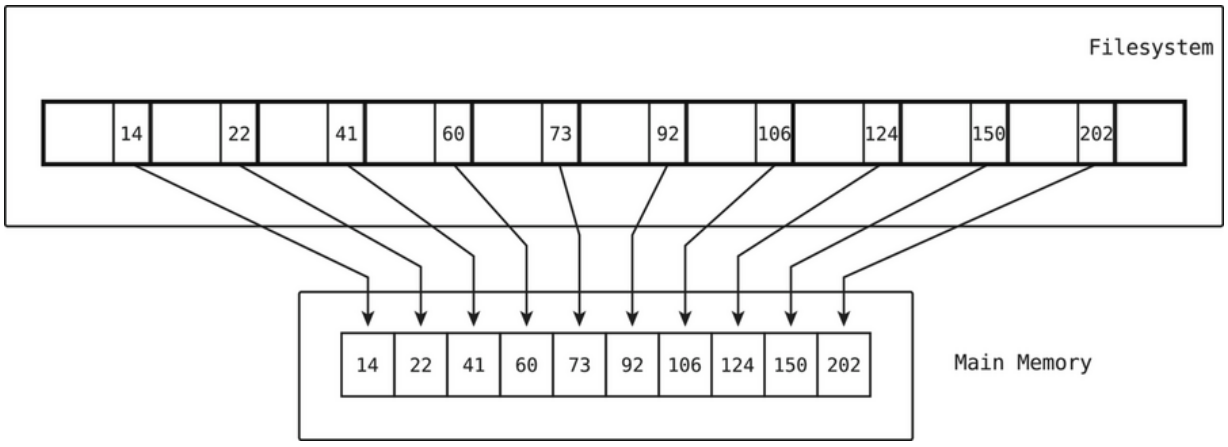
But, if the value we're searching for is *not* inside that block, then we only care about the first and last values of that block. By knowing those two values alone, we can determine whether the value we're searching for is somewhere to the left of this block, or if it's somewhere to the right of this block. So if our block size is 10, there are 8 elements of data that we don't need, as we only need the first and last elements. In real life, a block may hold *hundreds* of pieces of data, so we've kind of wasted an I/O to transfer a whole lot of useless data.

To optimize this algorithm, we need to consider whether we can pack the block with more useful information—and the more, the better. It's kind of like hiring a moving truck; we want to fill every cubic inch of that truck with as much stuff as we can. We don't want to have to make extra trips if we can avoid it.

Indeed, in our case of searching data from an ordered list, there's a theoretical way to pack a block of data with much more useful information. Here's how.

Packing Blocks

Instead of performing binary search and using up an I/O to transfer the centermost block of our list into main memory, we can transfer a block that is made up of information drawn from *across* our list. Specifically, we'll place into the block every 10th element from the list (except for the final value), like so:

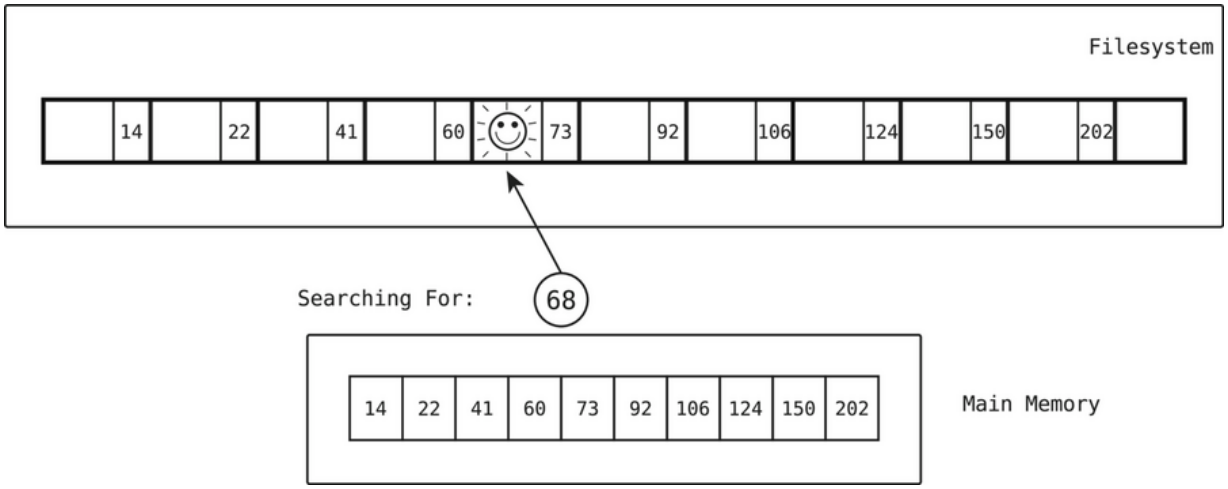


Here, we've created a block of 10 elements from data evenly interspersed across the original list.

In other words, instead of using an I/O to transfer a block of contiguous data from the file, we use that I/O to transfer a block of data made up of every 10th value in the list.

Now, here's the clincher. We've used just one I/O so far, but *we only need one more I/O to find our desired value*.

For example, let's say that we're searching for the integer 68. We'd look at the block we've loaded into main memory, and see that 68 would lie between the fourth value (the 60) and the fifth value (the 73). With this information, we'd know for certain that the 68 must be contained within the fifth block of the list as shown in the figure.



At this point, we only need to perform one more I/O to obtain the block where the 68 lives! We've successfully devised an algorithm that searches the ordered list in a maximum of 2 I/Os.

Now, the difference between 2 I/Os vs. the 4 I/Os may not seem like much. But in real life, when dealing with big data, our optimization can make a big difference.

For example, suppose we have a computer with a block size of 1,000. If we had a list of one billion elements and we make it so that our first I/O is a block comprised of every 1,000th element from the original list, we effectively break down the original list into about 1,000 sections, each of which contains one million elements. (Technically, it's 1,001 sections.)

Note that in this case, unlike the previous example, these 1,000 sections aren't equivalent to 1,000 blocks. For when we break up one billion elements into 1,000 sections, each section contains *one million* elements. And our block size is only 1,000.

That being said, after this first I/O, we can already pinpoint in which section our value lives. Again, this section contains one million elements, and our block size is 1,000, so we need to dig deeper.

We then take that section of one million values and create a second I/O comprising every 1,000th element, breaking up this section into 1,000 smaller sections, each of which contains 1,000 elements. At this point, each section is indeed the same size as a block.

By analyzing the block currently loaded into memory, we can now locate in which 1,000-element section (out of the one million elements that are in our search space) our desired value lives.

We can then use a third and final I/O to load that 1,000-element block into RAM and locate the desired value.

So, we only need 3 I/Os to find a value in a list of one billion elements! With our first approach of unoptimized external-memory binary search, this would have taken up to *20* I/Os.

For now, I'll refer to this clever new algorithm as "Optimized Search." In truth, it doesn't have a name because it's not used in real life for reasons you'll see soon, but let's run with it for now.

Back to Big O

We've seen that the speed of unoptimized external-memory binary search is O(logN/B) I/Os. As we'll see soon, the Big O of *Optimized* Search is also, technically, O(logN/B) I/Os. But because both algorithms have the same Big O speed, it turns out that Big O doesn't capture the performance gains of Optimized Search over its unoptimized counterpart.

If we want to articulate the speed of Optimized Search to show to what extent it's faster than unoptimized external-memory binary search, we'd have to use a more fine-grained approach than Big O. And to do this, we first need to change up the way we've been expressing logarithms.

Throughout this book, we've been referring to "log N" without specifying the logarithm's base. This is because Big O notation doesn't care about the

base, as the base is considered a constant.

However, in practice, the base will make a big difference in the context of our discussion of external-memory algorithms. So, in this chapter (as well as the next), I will make a point to specify a logarithm's base.

With unoptimized external-memory binary search, we'd say that it performs $\log_2 N/B$ I/Os. As I explained in Volume 1, Chapter 3, the way I like to think about this logarithm is: how many times do I need to divide a number by 2 until I end up with a result of 1? For example, $\log_2 1{,}024$ is 10 since I have to divide 1,024 10 times until I end up with 1.

The question to consider now, though, is the precise number of I/Os that Optimized Search executes.

With Optimized Search, the way we cut down the list of blocks depends on the block size. If the block size is 10, for example, each I/O divides the original list into 11 sections. Each subsequent I/O takes one of those 11 sections and divides it into a list of 11 smaller subsections. We'd therefore say that this takes $\log_{11} N$ I/Os. Put another way: this is the number of times we need to divide the list by 11 until we get a result of 1.

If, however, the block size is *1,000*, then each I/O reduces the original list into 1,001 small subsections. Accordingly, we'd say that it takes $\log_{1,001} N$ I/Os to find the value we're looking for. Indeed, that's why when we have a list of one billion elements, it takes just 3 I/Os to perform Optimized Search. That is, $\log_{1,001}$ of one billion is approximately 3.

It emerges that Optimized Search takes roughly $log_B N$ *I/Os* for N data elements. That is, we keep dividing a list of N elements by the block size until we find our search value.

If we wanted to express this even more accurately, we'd take note of the fact that when we divide a list by B, we produce B + 1 sections.

To wrap this nicely in a Big O bow, we'd say that the speed of Optimized Search is $O(\log_{B+1} N)$. As you've seen, this is much faster than $O(\log_2 N)$. In fact, the greater B is, the faster the algorithm.

The Impracticality of Optimized Search

While the Optimized Search algorithm seems great at face value, it can't be implemented in reality—at least not in the particular way I've described it. The entire premise of Optimized Search is that we tell the computer to create a block using elements scattered across the computer's memory. However, there's a problem with this approach.

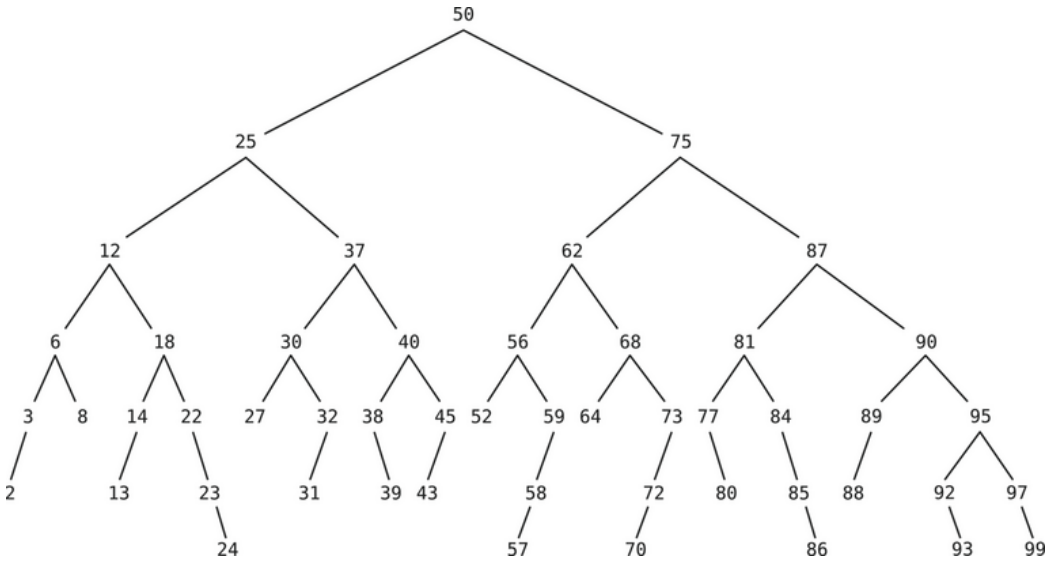
The thing is, when a computer performs an I/O, we don't get to tell it how to form its block; it does so automatically. And when a computer creates a block, it only does so using a chunk of contiguous memory. We simply don't get to tell the computer to form a block out of elements interspersed throughout the data.

But there's good news. While this version of Optimized Search may not be practical, there's a data structure out there that operates on the same principles. This data structure uses a similar approach to Optimized Search and drastically reduces the number of I/Os needed for external-memory search. It's no surprise that this data structure is popular.

Binary Search Trees in External Memory

Before revealing this exciting new data structure, let's first talk about binary search trees (BSTs) and how they might work with external memory. All of our discussions about BSTs until this point assumed that the entire tree would fit into RAM. But what would it look like if the BST was too large for RAM? This could happen if our BST was absolutely huge or if our computer had a tiny main memory, or both.

Say, for example, that we have a microcomputer that has a main memory that could only hold up to 4 nodes of a BST. Let's also say that the block size was 4 nodes as well. How could our computer perform a search on the following BST, which contains 50 values?

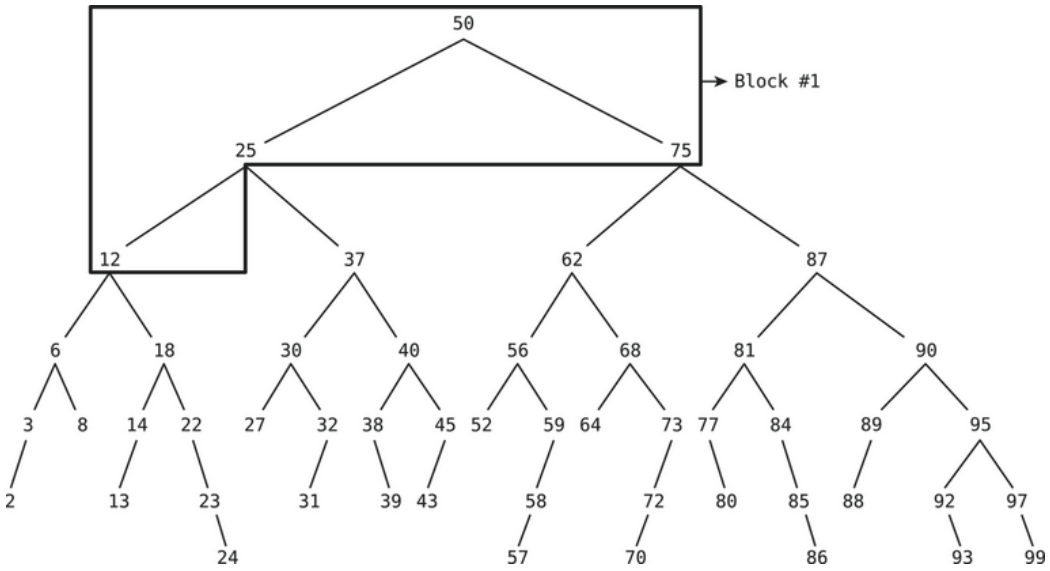


Well, first, this BST would need to be stored in the filesystem since that's the only place where we could even fit all this data. Let's assume, then, that we have a file that stores all of these nodes. How might the computer perform a search on this BST?

Let's apply our newfound knowledge about external-memory algorithms and walk through the steps the computer would perform if it were to search for the value 93 in the BST.

We begin our search at the root. Because the tree is currently in the filesystem, the computer needs to perform an I/O to transfer data to RAM. Because the computer's block size is 4, we can transfer a block of up to 4 nodes.

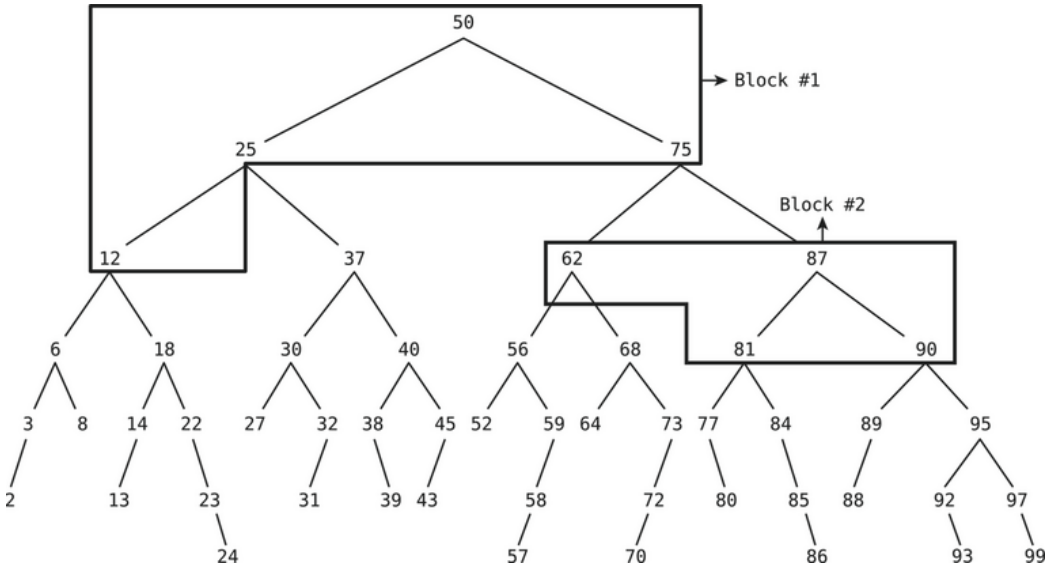
Now, it's hard to know exactly which nodes the computer might grab unless we can see exactly how the data is organized in the file. Let's make the optimistic assumption that the computer will grab nodes that are in close proximity to each other. Accordingly, the computer might transfer this block of 4 nodes when it grabs the root as shown in the figure.



This is our first I/O.

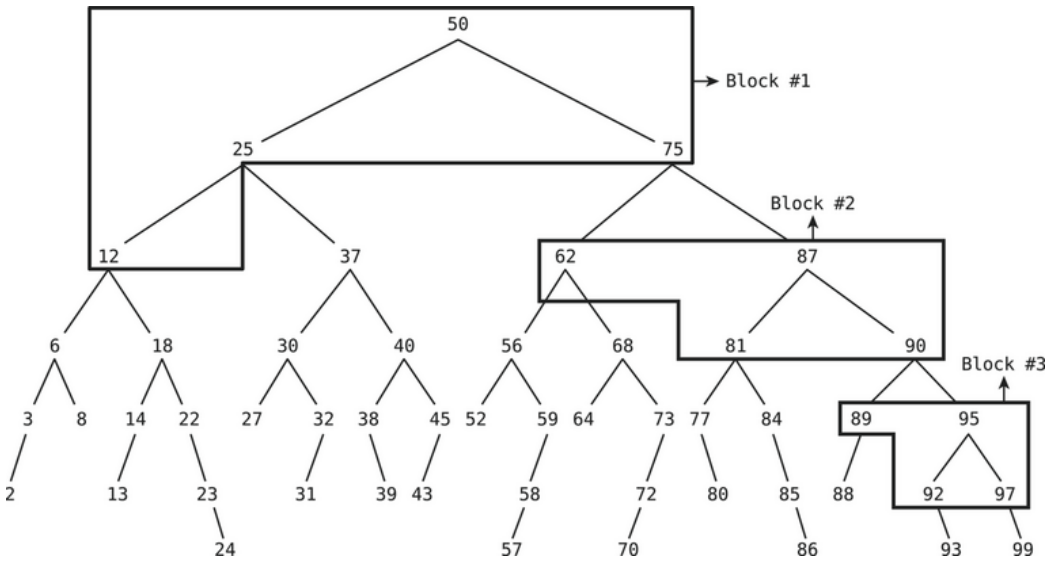
With the first block now in memory, the computer begins its search. We're searching for 93, which is greater than the root of 50, so we move to the root's right child.

Luckily, the right child, which is the 75, is already in main memory. Now, 93 is greater than 75, so we need to move to the 75's right child. However, we don't have that child in memory yet, so we need to perform our second I/O to transfer the next block of nodes:



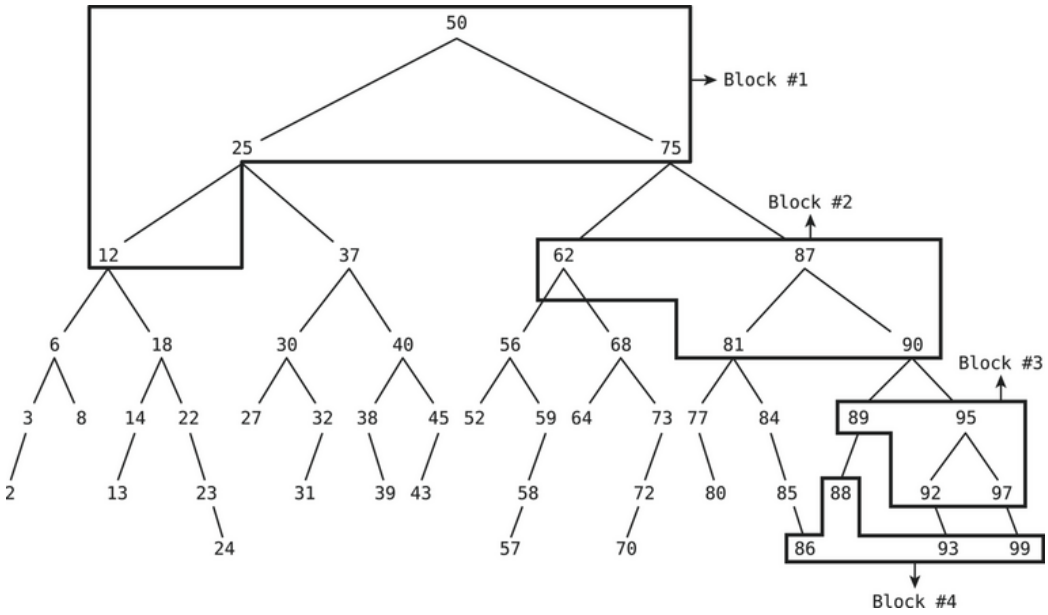
Again, we don't know for sure if this is the exact block the computer will transfer; it's just an example.

We now have the 75's right child in memory, which is the 87. 93 is greater than 87, so we move to the 87's right child, the 90. Our search value of 93 is greater than 90, so we need to move to the 90's right child. However, it's not in memory, so we need to perform our third I/O as shown in the figure.



With this third block in memory, we now have access to the 90's right child, which is 95. Our search value of 93 is *less* than 95, so we move to the 95's left child, 92.

Because 93 is greater than 92, we need to move to the 92's right child. It's not in memory, so we perform our fourth I/O:



Finally, we find the 93 we've been looking for. All in all, it took us 4 I/Os. Naturally, a much larger tree might require many more I/Os.

Optimizing External-Memory Trees

How might we optimize our tree to allow for faster search? Again, let's review the key to optimizing external-memory algorithms: we should aim to pack as much useful information as we can inside each block. With that in mind, let's ask ourselves whether the blocks from the previous BST example contained useful information.

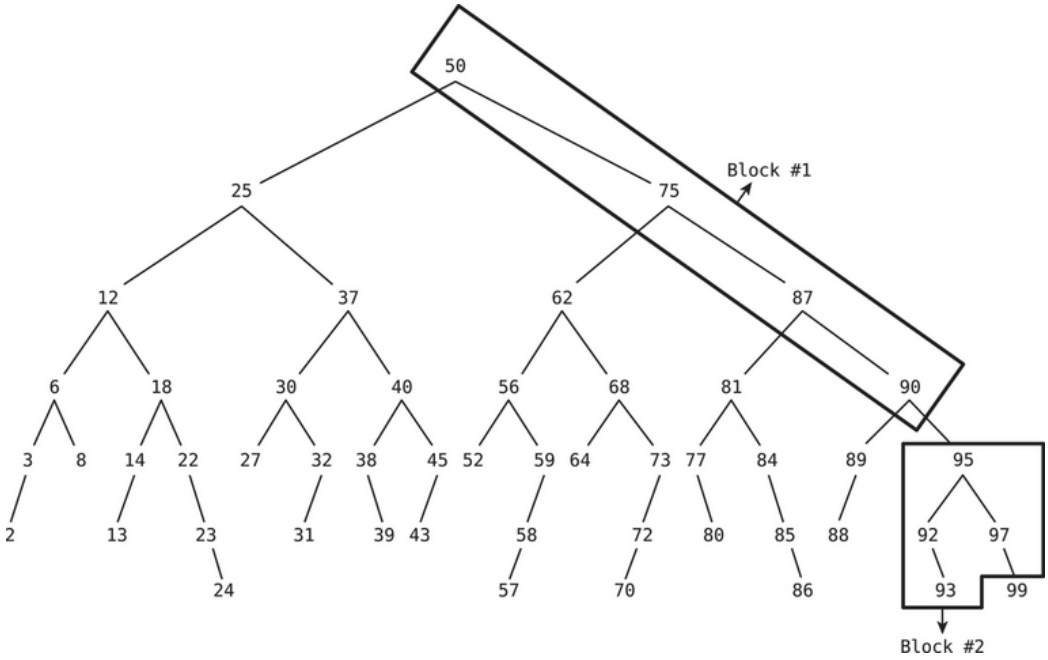
Looking at the particular blocks the computer transferred, each block ended up containing two useful values. The first block, for example, contained the root and the 75. We needed both of those values since that's the beginning of the path down to the 93. However, the block *also* contained information that we did *not* need, since the 25 and 12 were completely useless to us.

Similarly, the second block contained the 87 and 90 that we needed, but a 62 and 81 that we didn't. The same is true for the third and fourth blocks, as

at least half of each block contained useless data.

To optimize external-memory tree search, we need to try to pack blocks with more useful information. In a perfect world, *all* of a block's data should be useful to us.

In the BST shown earlier, it would have been nice if the first block contained the 50, 75, 87, and 90. And if the second block contained the 95, 92, and 93, we could have found the 93 in two I/Os:



But again, there's no way we can ensure that the computer will choose these blocks. We could, theoretically, structure the file's data in such a way that the 50, 75, 87, and 90 are all adjacent to each other so they end up in the same block. However, this only optimizes for values descended from the 90, and would end up making *other* values even slower to find. Imagine that we were searching for the 12, for example. If the root's block also contained 75, 87, and 90, we wouldn't even have transferred the root's left child of 25 into memory, which is what we need to find the 12.

Our goal is to create a different kind of tree that somehow reduces I/Os by packing each block with useful data. Well, it turns out that computer

scientists have already figured that out.

Enter the B-tree.

B-Trees

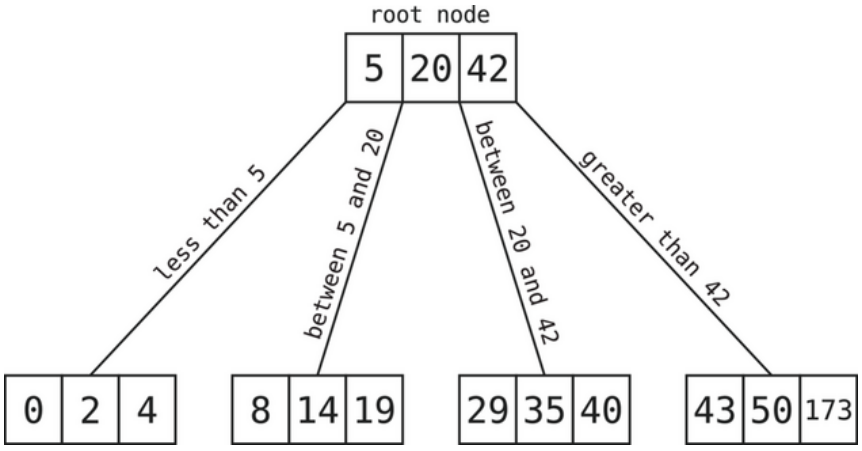
A *B-tree* is a tree data structure designed especially for external memory, and aims to keep I/Os to a minimum. In a bit, I'll get into all of the B-tree rules and operations, but let's begin with an overview of how it works.

The first thing to know about B-trees is that they are *not* binary trees. A binary tree is, by definition, a tree whose nodes have up to two children *at most*. Indeed, the trees we've dealt with so far in this book were types of binary trees.

A B-tree, on the other hand, contains nodes that can have *many* children. In fact, in the real world, B-tree nodes often have *hundreds* of children. Yes, you read that right. As we'll see shortly, the power of B-trees stems from this idea.

What is clear is that the "B" in the name "B-tree" does *not* stand for "binary." As to what it does stand for, well, there's no definitive answer to that. For whatever reason, the inventors of the B-tree never explained what the "B" represents. While various suggestions abound (including "bushy"!), the most we have to go on is the cryptic statement by one of the B-tree's inventors, who said, "the more you think about what the B in B-trees means, the better you understand B-trees." This is truly computer science at its best.

In addition to a B-tree node being able to have numerous children, each node can also hold numerous *values*. Here's an example of a small B-tree:



Here we can see that the root node has 3 values and 4 children. The pointers to the children live *in between* the actual values, and help us find which child we may be looking for. To find values less than 5, for instance, we need to follow the pointer to the left of the 5. To find values between 5 and 20, we need to follow the pointer that lives between the 5 and the 20. And so on.

When creating a B-tree, a decision needs to be made at the outset as to the maximum number of values each node can store. This decision will be applied to *all* nodes. In this example, each node stores a maximum of 3 values. In real life, it's not uncommon to have a B-tree whose nodes store a maximum of 500 values each. In this book, I'll refer to this number as a B-tree's *node maximum capacity*. (This is not official jargon. Official jargon uses the term "order." However, some computer scientists use the term "order" in slightly different ways, causing a fair amount of confusion. As such, I'm going to stay away from that term.)

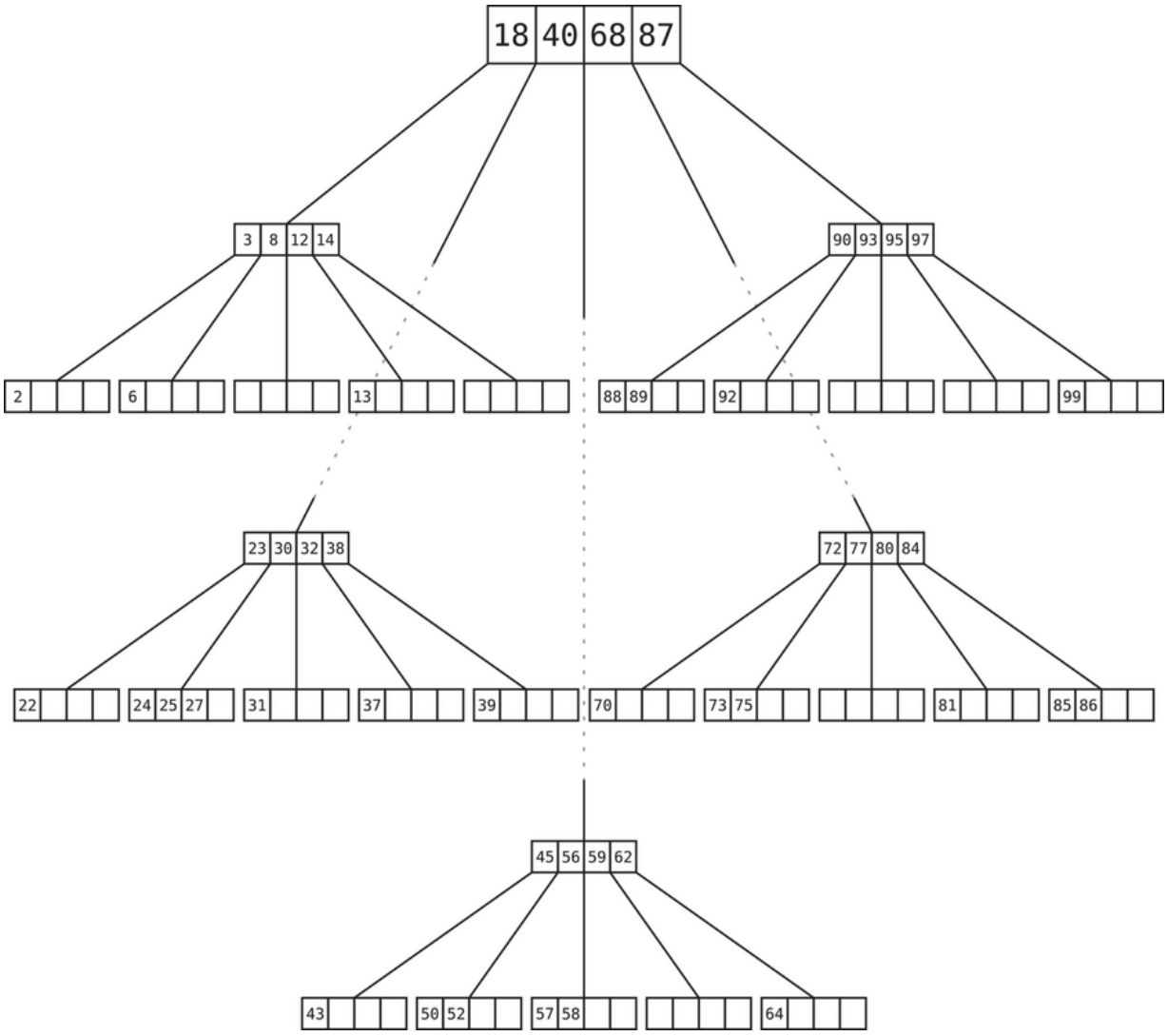
Note that the maximum number of children that a B-tree node can have is the node's maximum capacity *plus 1*. In the example tree, you can see that the node's maximum capacity is 3, but the maximum number of children each node can have is 4.

B-trees make use of the key technique we've been discussing to minimize I/Os, which is to pack as much useful data into a block as it can. When we create a B-tree, we can take advantage of this by doing something clever:

we make it so that a B-tree's node maximum capacity is the same as the computer's *block size*. So, if a computer's block size can hold 500 values (plus the pointers to the children nodes), we would decide that our B-tree's node maximum capacity should be 500. Let me explain why this is advantageous.

A B-tree's data is stored in the filesystem. Because each node fits perfectly into a block, we only need one I/O to transfer a node into RAM. To see how this will help us reduce I/Os, let's take the same 50 values from our example BST and store them in a B-tree. Let's say that we're still working with our tiny computer that has a block size of 4. This means that we can store 4 values plus 5 pointers to children.

The B-tree shown is what it may look like. I compacted the width of the tree so it can fit on the page.



As an example, suppose we are searching for the value 39. (Can you spot it?)

Recall that in our BST, it took 4 I/Os to find our desired value. Let's see how many I/Os it takes to find the 39 inside our B-tree.

First, our computer needs to access the B-tree's root node, which takes 1 I/O. The root node contains the values 18, 40, 68, and 87.

Because 39 is between 18 and 40, we follow the pointer that lives between 18 and 40 to find the next node. This is the node containing the values 23, 30, 32, and 38. We spend our *second* I/O loading this node into memory.

Now, 39 is greater than 38. Therefore, we need to follow the pointer that lies to the right of the 38, which leads to another node.

And so, we spend our *third* I/O loading this child. And whaddya know? The node contains the 39 we've been looking for!

In total, we only had to execute 3 I/Os to find our value. Now, this may only seem like a small savings compared with the 4 I/Os of the BST, but I need to point out several important things.

First, with the BST, we only *surmised* that it might take 4 I/Os. It's possible that the data of the BST might be structured in a way that might take more I/Os. With our example B-tree, though, we *guarantee* that we can find any value in 3 I/Os.

Second, note that we've not filled the leaf nodes to capacity. We simply took the same 50 values from the BST and used them to populate a B-tree. This B-tree, though, has the capacity to store up to *124* values. (That's 4 values at the root level, 20 values at the second level, and 100 values at the bottom level.) And even with 124 values, we *still* guarantee that we can find any value with 3 I/Os. The BST, on the other hand, could certainly take many additional I/Os if it held 124 values.

Third, I've used a B-tree with a node maximum capacity of 4 only because otherwise my diagram would be too large to fit on the page. In practice, B-trees often have a node maximum capacity of *1,000*. This means that such a B-tree could fit one *billion* values on 3 levels. (1,000 * 1,000, * 1,000 = 1 billion.) And that means that we *still* only need 3 I/Os to find a value in such a tree!

Why B-Trees Work

Recall that the general approach to optimizing external-memory algorithms is to pack a block with useful data. Now, when it comes to B-trees, each block contains a single node. For example, if our B-tree's nodes each

contain 1,000 values, these 1,000 values are *all* useful because they *guarantee that the node's children values will be divided into 1,001 subsections*.

Because we will only select one of those subsections, our search space has now become about *one 1,000th* of its original size. So, if the entire tree has one billion values, after our first I/O, we now only have one million values to search from. And after we perform our second I/O and load our next node, we use that node's values to split the remaining search space into 1,001 subsections, each of which has 1,000 values. So, after two I/Os, we've reduced our total search space down to 1,000 values.

This is the same general technique we used with Optimized Search. That is, both Optimized Search and B-trees pack a block with values interspersed across the whole data set, and these values split the data set into many small subsections. Each subsection, once accessed, divides the remaining search space into even smaller subsections, and so on.

B-Tree Efficiency

Because searching a B-tree is so similar to Optimized Search, the way we describe the time complexity for both algorithms is the same.

When we search a B-tree that has a node maximum capacity of 1,000, we end up performing $\log_{1,001} N$ I/Os. That is, each I/O divides the search space by 1,001. We keep doing this until we find the one singular value we're looking for. Similarly, if the node maximum capacity is 4, we perform $\log_5 N$ I/Os.

Essentially, the logarithm base is determined by the node maximum capacity, which is also equal to the computer's block size. As with Optimized Search, the logarithm base is the block size plus 1. In other words, we perform $\log_{B+1} N$ I/Os.

This is potentially way faster than searching a BST, where in a worst-case scenario, each I/O only includes one useful value. This means that each I/O moves us one child down the BST, which in turn reduces the remaining search space by half. This would be described as $O(\log_2 N)$ I/Os.

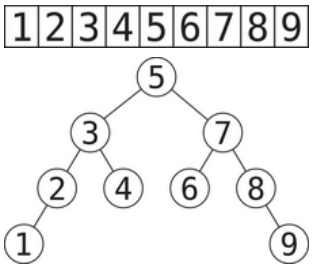
As to how much faster B-trees are than BSTs depends on the block size. Even if a B-tree has a block size of 2, its speed of $\log_3 N$ would still be considerably faster than a BST's speed of $\log_2 N$. And as a B-tree's block size increases, its search speed becomes faster and faster, leaving BSTs in the dust.

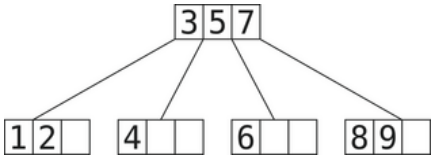
Again, I'll note that in terms of pure Big O Notation, we drop the base of the algorithm, so B-tree search and BST search are technically both O(log N). However, this is another example of where Big O Notation is limited in helping us conceptualize the true difference between two competing algorithms.

B-Trees In Main Memory

We've seen how B-trees surpass BST speeds when it comes to external memory. But let's say that our data is small enough to fit inside main memory. Might we still want to reach for B-trees instead of BSTs? Let's do a quick analysis. For argument's sake, let's assume we're dealing with trees that are perfectly balanced.

Here are three in-memory data structures that all hold the numbers 1 through 9 in order. We have an ordered array, a BST, and a B-tree:





Say that we're searching for the value 4. With each data structure, we'd perform some type of binary search. Let's walk through the steps simultaneously with all three data structures.

Step 1: With the ordered array, we perform classic binary search. This means we access the centermost value, which is the 5.

Likewise, in the BST, we access the root first, which is also the 5.

In the B-tree, we begin with the centermost value of the root node, which—you guessed it—is the 5.

Step 2: Because the 4 we're looking for is less than 5, we must turn toward the left. With the array, we pick one of the center items in the left half of the array—say, the 3.

In the BST, we access the 5's left child, which is the 3.

With the B-tree, we perform binary search *on the root node*, turning toward the left half of the node. In this case, the only item to the 5's left is the 3.

Step 3: Because 4 is greater than 3, we must now look to the right of the 3 (but still left of the 5). In the ordered array, we find our 4 at this point.

Similarly, in the BST, we turn to the 3's right child, which is the 4.

And with the B-tree, because there are no more values to choose from in the root node, we follow the pointer between the 3 and the 5. This leads, of course, to the 4.

It turns out that the search took the same number of steps *with all three data structures*. But let me remind you why we'd choose one data structure over

the other, starting by comparing the ordered array with the BST.

While both data structures offer $\log_2 N$ search, a BST *also* offers $\log_2 N$ insertion and deletion. An array, on the other hand, can take as much as O(N) time for insertion. That is, if we insert or delete the left-most value of the array, we have to shift the remaining values of the array to either the right or the left.

If you don't need fast insertions or deletions, though, an ordered array is simpler and therefore easier to implement. But if you do need insertions and deletions, a BST is faster overall than an ordered array.

Now, a B-tree is kind of a hybrid between the ordered array and the BST. It's like a BST in that it's a tree, but like an array in that it can hold multiple values in each node. (With this perspective, we can look at an ordered array as one giant node.)

So, a B-tree isn't a great choice for in-memory algorithms, because it has drawbacks similar to the ordered array. That is, a B-tree will be slower than a BST when it comes to insertion and deletion. This is because a B-tree node holds multiple values like an array. So, if we insert a new left-most value into a B-tree node, we have to shift the remaining values to the right. Similarly, if we delete the left-most value, we'd have to then shift the remaining values to the left.

At the same time, if we don't need our data structure to implement insertion and deletion, we may as well use an ordered array instead of a complicated B-tree.

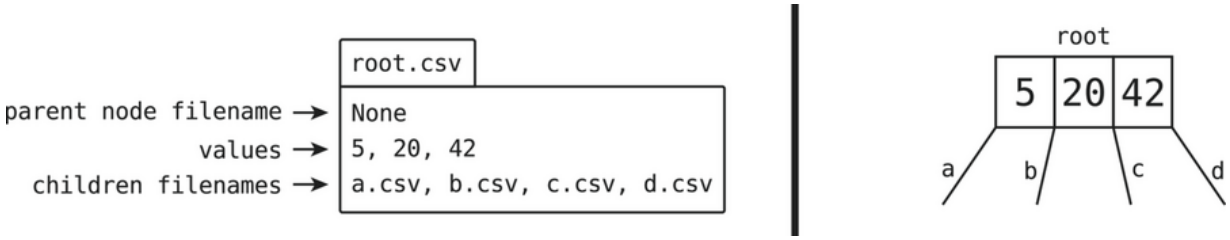
So, when it comes to in-memory data structures, the B-tree usually isn't your friend. But when it comes to *external* memory, the B-tree is a reliable pal.

Implementing B-Trees

There are numerous ways to implement a B-tree. Here, I've chosen a simple approach that will allow us to focus on the big picture of how B-trees work while avoiding getting into the weeds of optimizations and other hacks.

Let's begin by implementing a B-tree *node*. To mimic real life, I'll create nodes that will force the computer to perform an I/O to access the node. Specifically, I'm going to *store each node in a separate file*.

Our example node file is called `root.csv`. On the right side, you'll see a visual of the node we're representing with this file. On the left side, you'll see the file itself:

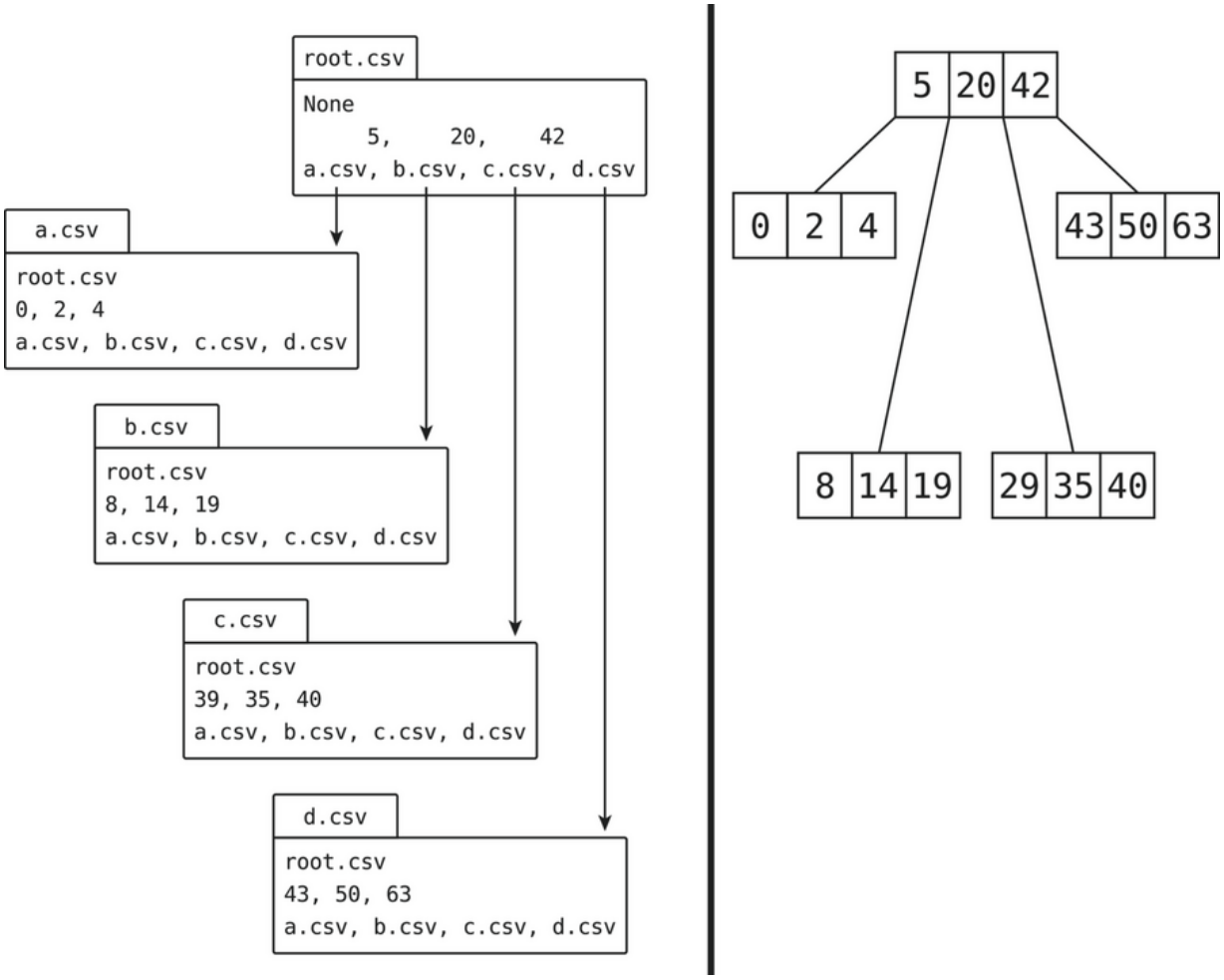


There are many ways I could have chosen to store the data in the file, but this was my arbitrary choice.

Each file stores a node's data using comma-separated values plus newlines. Accordingly, I've called the file extension `.csv`, which is a common convention for such files.

On the first line of the file, we store the filename of the node's parent. In this example, the file represents the root node, so it doesn't have a parent, and that's why the word `None` appears in the first line. The second line stores the node's values, and the third line stores the filenames of the node's children.

Here is a complete set of files representing an entire B-tree. To make the visual clearer, in the root.csv file, I added spacing around the values 5, 20, and 42, but in reality, those extra spaces will not exist:



Because each node is stored in a file, I'm not going to bother writing *code* to represent a node. All the information we need is already stored in the file itself!

Code Implementation: B-Tree Search

Let's continue our B-tree implementation by creating the search functionality. There's a fair bit of code here, but I'll break it down:

```
import os
```

```
class BTree:
    def __init__(self, root=None):
        self.root = root
        self.max_node_size = 4

    def search(self, search_value, node=None):
        node_file = node or self.root
        node_data = self.read_node_file(node_file)
        values = node_data.get('values')
        children = node_data.get('children')

        index = 0
        while index < len(values):
            current_value = values[index]
            if current_value == search_value:
                return [True, node_file, index]
            if search_value < current_value:
                if children:
                    child_to_follow = children[index]
                    break
                else:
                    return [False, node_file, index]
            index += 1

        # if search_value is greater than all values:
        if search_value > current_value:
            if children:
                child_to_follow = children[len(children) - 1]
            else:
                return [False, node_file, index]

        return self.search(search_value, child_to_follow)

    def read_node_file(self, node_file):
        with open(node_file, 'r') as reader:
            parent = reader.readline().rstrip('\n')
            values = reader.readline().rstrip(',\n').split(',')
            values = list(map(lambda x: int(x), values))
            children = reader.readline().rstrip(',\n').split(',')
            if children[0] == '':
                children = None

        return {'parent': parent, 'values': values, 'children': children}
```

Note that at the top of the file, we import the `os` Python module. We won't be using it for our `search` method, but we'll need it when we implement B-tree insertion later.

We kick off our `BTree` class like this:

```
class BTree:
    def __init__(self, root=None):
        self.root = root
```

The only class variable we keep track of is the file representing the `root` of the tree. When we create a brand-new empty tree, this will be `None` since we won't create the file until we begin adding actual data to the tree. But let's assume that when we call the `search` method, our tree already consists of multiple files.

Our `search` method begins as follows:

```
def search(self, search_value, node=None):
    node_file = node or self.root
    node_data = self.read_node_file(node_file)
    values = node_data.get('values')
    children = node_data.get('children')
```

The `search` method expects a `search_value` and a `node`, which will be a string representing the filename of one of the tree's nodes. The search will begin from that `node`.

The method begins by creating a variable called `node_file` which, at the beginning of a search, will point to the tree's `root`. Later, we'll recursively call the `search` method on children nodes, in which case the `node_file` will point to whichever child node we're searching next.

We then call a helper method called `read_node_file`, which reads the `node_file` and retrieves all of its data and stores it in the `node_data` variable. The data is stored as a hash table and contains the keys `values` and `children`. (It also

contains `parent`, but we're not going to use that right now.) We'll analyze the `read_node_file` method soon, but for now, let's continue to plow forward.

The next section is a loop that compares our `search_value` to each of the values in the node:

```
index = 0
while index < len(values):
    current_value = values[index]
    if current_value == search_value:
        return [True, node_file, index]
```

We create an `index` variable which starts at 0, and use this variable to retrieve each value of the node using `values[index]`. As we iterate, each subsequent value of the node becomes the `current_value`.

If `current_value == search_value`, meaning that we found our `search_value` in the node, we happily return the information regarding our find. I've chosen to return an array containing three values. The first contains `True` to indicate that the `search_value` is contained in the tree. (We return `False` if the `search_value` is *not* there.) Additionally, we return the `node_file` where the `search_value` can be found and the `index` pointing to the exact spot within the node where the `search_value` is located. Depending on what you're using a B-tree for, you may want to return other information.

If the current node does *not* contain the `search_value`, we continue our loop:

```
if search_value < current_value:
    if children:
        child_to_follow = children[index]
        break
    else:
        return [False, node_file, index]
index += 1
```

As we compare the `search_value` to each value in the node (the `current_value`), we check to see if the `search_value` is less than the `current_value`. If it is, and the current node has children, we want to follow the child pointer that is found immediately to the "left" of the `current_value`. This ends up being `children[index]`. That is, if we're up to, for example, the third value in the node, the third child in the node will be that value's "left" child. We store the child's filename in a variable called `child_to_follow`. (At the end of our method, we'll recursively call the `search` method on that child.) We also terminate the loop early at this point since we've already found the child we're looking for.

If, however, this node does not contain children, which is the case for leaf nodes, it must mean that the `search_value` is not present in the tree. Accordingly, we return an array whose first value is `False` to indicate this. Additionally, we return the `node_file` and `index` to represent the spot where the `search_value` *should* go if it were to exist. This information will be useful when we insert a new value into the B-tree.

Finally, we increment `index` and start the next round of the loop, which will continue until the index moves beyond all the values we have in the current node.

The next bit of code occurs when we reach the end of the current node without finding the `search_value`:

```
    if search_value > current_value:
        if children:
            child_to_follow = children[len(children) - 1]
        else:
            return [False, node_file, index]
```

We check whether the `search_value` is greater than the `current_value`. Because this code occurs after the loop has been terminated, the `search_value` being

greater than the `current_value` can only happen if the `search_value` is greater than *all* the values in the array.

Because the `search_value` is greater than all the values in the current node, we now have two possible paths. If the current node has no children, this means that the `search_value` is simply not present in the tree, so we include `False` in the array that we return.

But if the final value in the current node has a "right" child pointer, the corresponding child is the node we need to traverse next, so we assign that node to the `child_to_follow` variable. Again, the right-most child of the current node will contain values that are greater than the current node's right-most value. So, if the current node's final value is 56, and we're searching for a 73, we need to move on to the current node's right-most child.

Finally, our method concludes with the following line:

```
return self.search(search_value, child_to_follow)
```

That is, we recursively call this `search` method on whichever child node we have chosen to follow next.

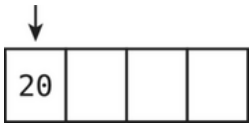
You may notice that I've chosen to perform a linear search on each node. That is, we use the `while` loop to sequentially compare the `search_value` with each value of the current node. We could, alternatively, have performed a *binary* search on each node, which would be a faster approach. I chose the linear search approach to keep our code simpler. In any case, don't forget the main idea I've been emphasizing throughout this chapter: when dealing with external memory, our primary focus should be on the number of I/Os that occur and not on the in-memory steps. Although performing binary search on the current node will reduce in-memory steps, it will not reduce the number of I/Os. I've aimed for simplicity instead.

B-Tree Insertion

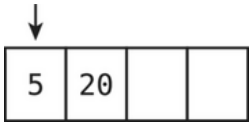
Like red-black trees and randomized treaps, B-trees are self-balancing. However, B-trees perform a different *kind* of self-balancing, as you'll now see.

In the following example, we're going to work with a B-tree whose nodes hold a maximum of 4 values. We'll assume that our B-tree is currently empty, so we'll start by inserting integers into the tree.

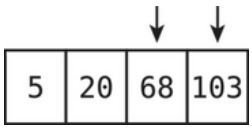
Step 1: Let's insert a 20:



Step 2: Next, we'll insert a 5. Because B-trees always hold their values in order, we'll need to move the 20 one slot to the right to make room for the 5. Here's what this looks like when we complete this:



Steps 3 and 4: We'll add the values 68 and 103 to the tree:



Now the party begins. Say that we want to insert a 150. It turns out that our node is already full, so here's what we'll do. I'll describe the algorithm first, and then show it visually so you can understand what I'm talking about.

A. Add the 150 to the current node *temporarily*, despite the fact that there will be too many values for the node to hold. (A node can hold just 4 values, and now there are 5.)

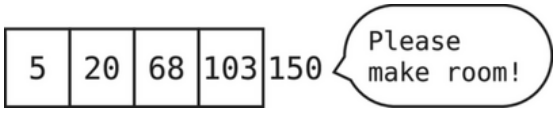
B. Grab the median value and remove it from the current node. The median value is the one in the center. For our example of 5, 20, 68, 103, and 150, the median value is 68.

C. Split the current node into *two* nodes, each of which contains half of the remaining elements.

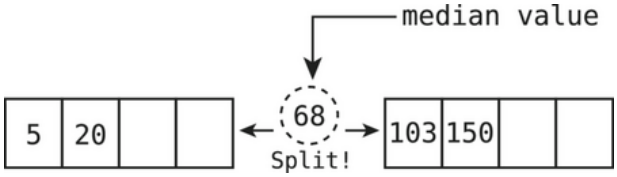
D. Move the median value into the parent node. If no parent node exists, create a *brand-new node* to house the median value. This new node will now be set as the parent to the two "split" nodes we created in the previous step.

Let's see now how this all plays out.

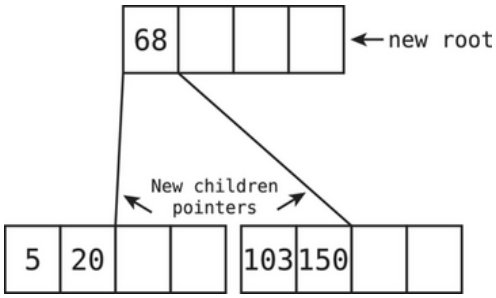
Step 5: Add the 150:



Step 6: Our node is now overstuffed, so we remove the median value (68) from the node, and split the current node into two:



Finally, we create a new node to house the median element and set it as the parent of the split nodes:

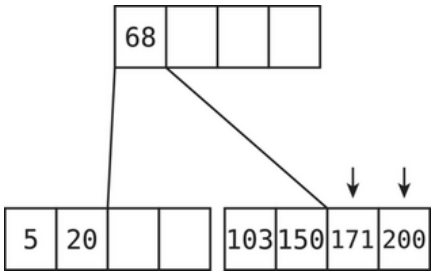


Note that the node with the 68 is the new root of our tree.

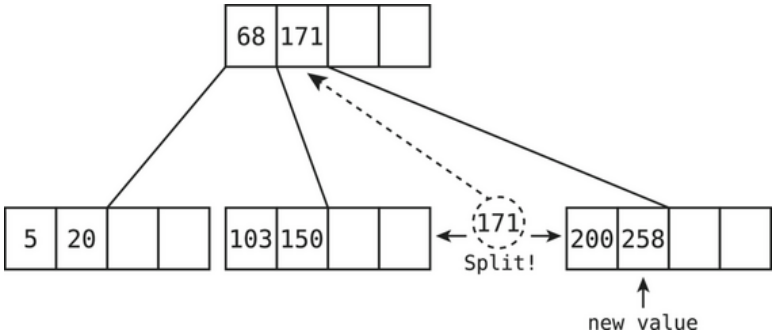
It can be said that a B-tree grows *upward*, as our original node was split into two and created a brand-new parent. This will become even more apparent as we continue with our walkthrough, so let's move on.

Now that we have a parent with children, it's time to introduce an important rule about B-tree insertion: *we always begin insertion by inserting the new value into a leaf node*. The new value may at some point work its way up the tree, but it always begins its life in a leaf node. Let's see how this plays out in the next steps.

Steps 7 and 8: Let's insert a 171 and 200. In *theory*, we could place them into the root node. But as you've just learned, we only insert into leaf nodes. So, we have to search the tree to find in which leaf node they belong. In our case, they belong in the right child:



Step 9: Next, we'd like to insert 258, which is a fine number indeed. However, there's no room for it in the right child, as shown in the figure. This means it's time to split!

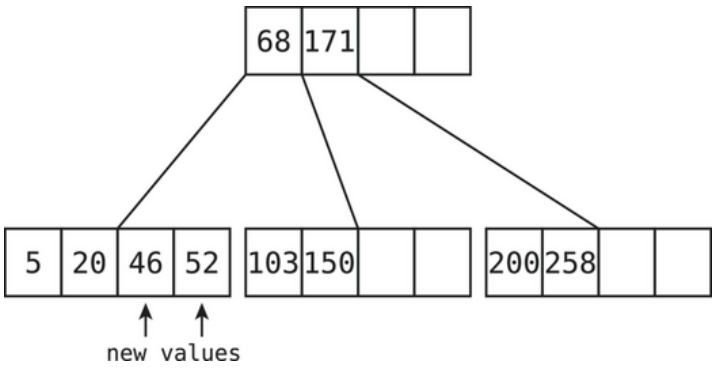


We split the right child into two nodes while grabbing the median value, the 171. In our previous split, we created a brand-new node for the median

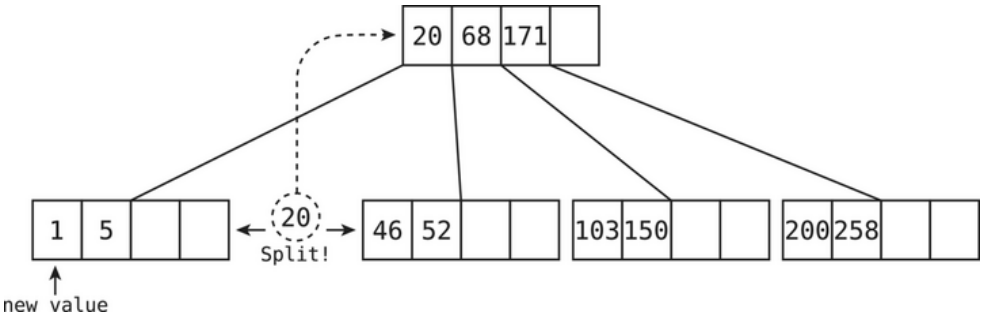
value. Now, however, because the 171 can fit nicely into the root node, we place it there.

Let's keep going. Until I say when, the following steps will repeat the same algorithm we've followed until now.

Steps 10 and 11: We'll insert the integers 46 and 52. Remember that we always insert into a leaf node:

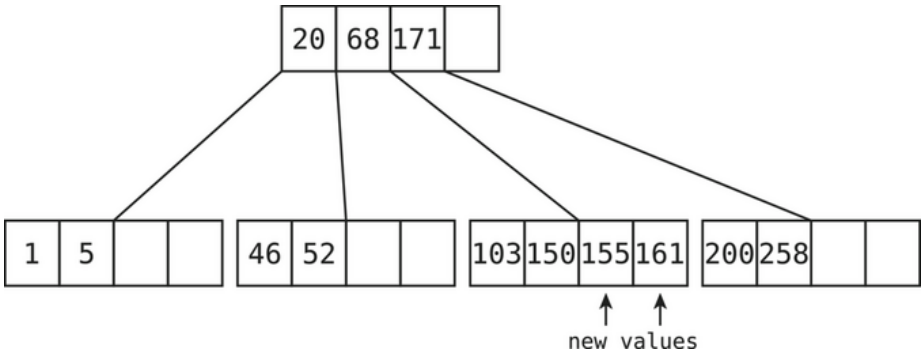


Step 12: Next, we'll insert a 1. It belongs in the left-most child, but it doesn't fit. That means we need to split:

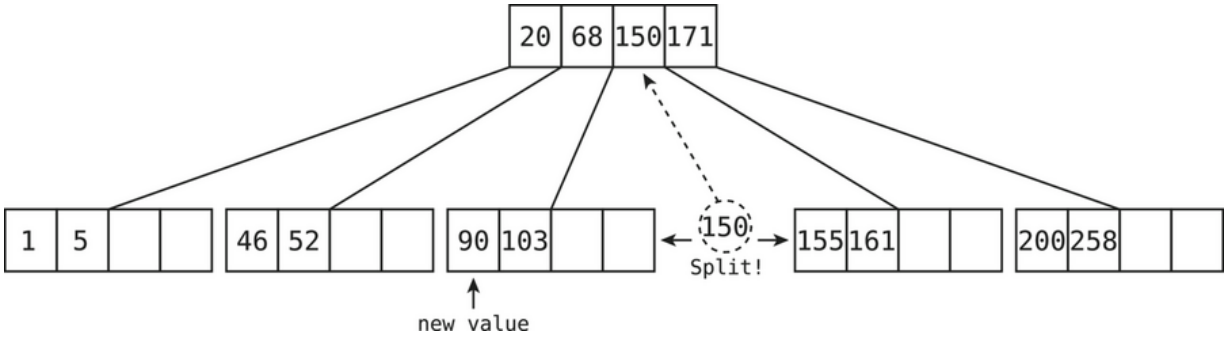


As you can see, the median value of 20 moves up to the root node, shoving the 68 and 171 over to the right to make room.

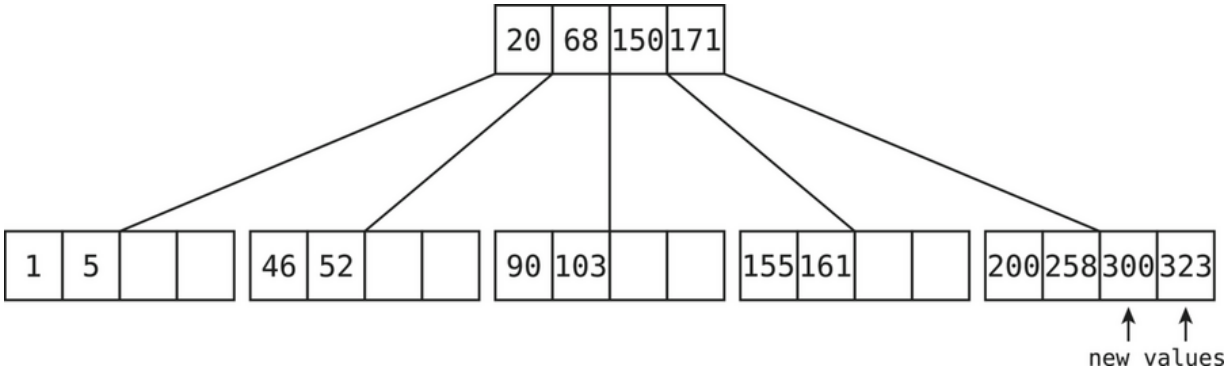
Steps 13 and 14: Insert a 155 and 161:



Step 15: We insert a 90, causing a split:



Steps 16 and 17: Insert a 300 and 323:

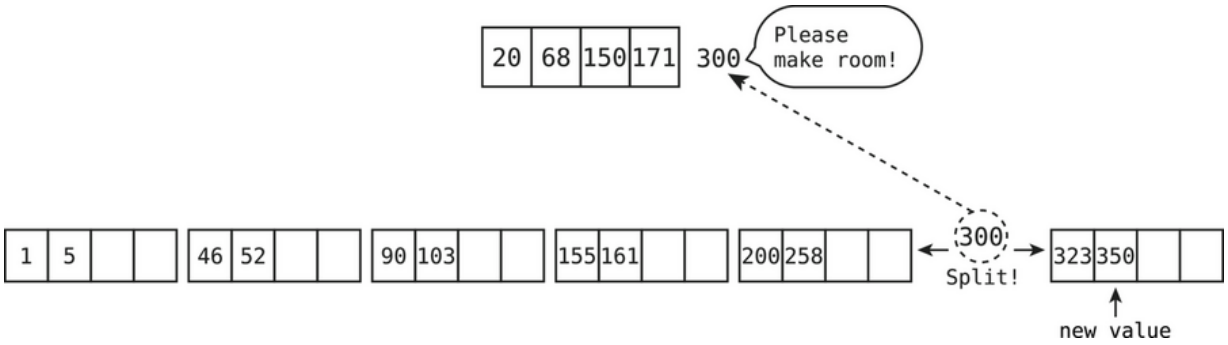


Okay, my friends, it's time. We're up to the grand finale. (I recommend listening to the climax of Tchaikovsky's 1812 Overture while reading the next steps.)

Step 18: We ever so innocently insert a 350, unwittingly setting off a chain reaction.

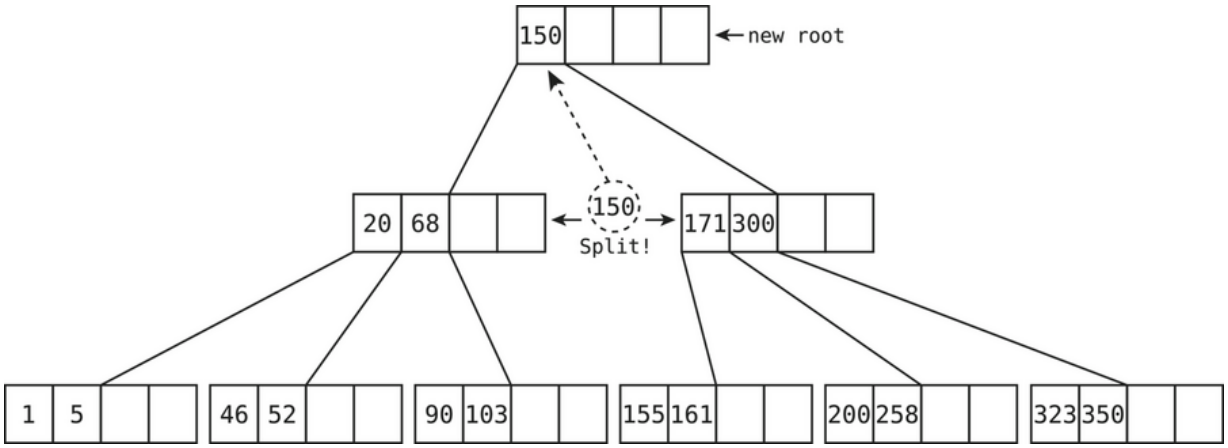
The 350 belongs in the right-most child, but there's no room. So, we split the child, and grab the median, 300. Now, we *try* to insert the 300 into the

root node:



But there's no room! Can you guess what happens next?

Well, I'll tell you. We split the *root node* in two, and grab *its* median, the 150. Because the root node has no parent, we create a brand-new root node that houses the 150 as shown in the figure.



In other words, whenever we split a node, we do so recursively. That is, we keep splitting nodes up the tree until we have no more nodes that are overstuffed.

I mentioned earlier that a B-tree grows upward, and this last insertion gives you a sense of that.

Well, we did it.

The B-Tree Insertion Algorithm

Let's recap the B-Tree insertion algorithm:

1. If the tree is empty, we create a node to house the inserted value.
2. We search the tree for the correct leaf node to insert the new value, and attempt to insert the value there.
3. If, after inserting the new value, the leaf node has too many values, we *recursively* both split the nodes and move the median up the tree. The specific details of this are outlined in Steps 4–6.
4. Remove the median value and split the leaf node into two nodes.
5. If the leaf node doesn't have a parent, we create a new node to house the median value. If the leaf node *does* have a parent, we insert the median value into the proper place within the parent.
6. If the parent now has too many values, this is where the recursion kicks in, and repeats Steps 4–6 on overstuffed nodes until there are no more nodes that have too many values.

Code Implementation: B-Tree Insertion

I'll admit that the code for B-Tree insertion isn't short. Don't feel guilty if you want to skip it. It's there for those who are interested in the nitty-gritty details.

If you're still here, I now present to you the insertion code in all its glory, after which I'll break it down:

```
    def insert(self, value):
        # if tree is empty:
        if not self.root:
            self.create_root(value)
            return

        search_result = self.search(value)
```

```
        # if value is already in tree:
        if search_result[0]:
            return

        node_file = search_result[1]
        self.insert_into_node(node_file, value)

    def insert_into_node(self, node_file, value):
        node_data = self.read_node_file(node_file)
        values = node_data.get('values')
        children = node_data.get('children')
        parent = node_data.get('parent')

        values.append(value)
        values.sort()

        if len(values) > self.max_node_size:
            self.split_node(node_file, parent, values, children)
        else:
            self.write_to_node_file(node_file, parent, values, children)

    def split_node(self, node_file, parent, values, children=None):
        os.remove(node_file)

        median_index = self.max_node_size // 2
        left_node_filename = str(values[0]) + '.csv'
        right_node_filename = str(values[median_index + 1]) + '.csv'

        if parent == 'None':
            parent_node_filename = str(values[median_index]) + '.csv'
        else:
            parent_node_filename = parent

        # Split the current node by creating two new nodes
        # (a left node and right node):
        if children:
            left_children = children[:median_index + 1]
            right_children = children[median_index + 1:]
        else:
            left_children = None
            right_children = None
```

```
        self.write_to_node_file(left_node_filename, parent_node_filename,
                                values[:median_index], left_children)

        self.write_to_node_file(right_node_filename, parent_node_filename,
                                values[median_index + 1:], right_children)

        if parent == 'None':
            new_parent = self.write_to_node_file(parent_node_filename,
              'None', [values[median_index]],
              [left_node_filename, right_node_filename])
            self.root = new_parent
        else:  # if current node has a parent:
            # We will soon split the current node into two new nodes, so we
            # have to add these nodes to the list of the parent's children:
            parent_data = self.read_node_file(parent_node_filename)
            index_of_node_file = parent_data.get('children').index(node_file)

            updated_children = parent_data.get('children')[:index_of_node_file]
+ \
              [left_node_filename, right_node_filename] + \
              parent_data.get('children')[index_of_node_file + 1:]
            self.write_to_node_file(parent_node_filename,
              parent_data.get('parent'),
              parent_data.get('values'),
              updated_children)

            # Insert the center value into the parent node:
            self.insert_into_node(parent_node_filename, values[median_index])

        # Update the left and right nodes' children to reflect their new
    parents:
        if left_children:
            for child in left_children:
                child_data = self.read_node_file(child)
                self.write_to_node_file(child, left_node_filename,
                  child_data.get('values'), child_data.get('children'))

        if right_children:
            for child in right_children:
                child_data = self.read_node_file(child)
                self.write_to_node_file(child, right_node_filename,
                  child_data.get('values'), child_data.get('children'))
```

```
    def write_to_node_file(self, node_file, parent, values, children=None):
        values_string = ''
        for value in values:
            values_string += str(value) + ','

        if children:
            children_string = ''
            for child in children:
                children_string += str(child) + ','

        with open(node_file, 'w') as writer:
            writer.write(parent + '\n')
            writer.write(values_string)
            if children:
                writer.write('\n' + children_string)

        return node_file

    def create_root(self, value):
        filename = 'root.csv'
        with open(filename, 'w') as writer:
            writer.write('None\n')
            writer.write(str(value) + ',')
        self.root = filename
```

Let's take it from the top, starting with the `insert` method. It accepts a `value` parameter, which is the value we're going to insert into our tree:

```
    def insert(self, value):
        if not self.root:
            self.create_root(value)
            return
```

First, if the tree is entirely empty and doesn't have any nodes yet, we create a root that will house the `value` we're inserting. To accomplish this, we rely on the helper method `create_root`, which you can find at the end of the code listing.

Here's the remainder of the `insert` method:

```
        search_result = self.search(value)
```

```
    if search_result[0]:
        return

    node_file = search_result[1]
    self.insert_into_node(node_file, value)
```

We call the `search` method from earlier in this chapter to find the `value` in the tree. Whether the `value` is in the tree or not, we get back an array with some important pieces of information. We store this array in a variable called `search_result`.

If we find that the first item in the `search_result` array is truthy, this means that the `value` we're trying to insert is already in the tree. If this is the case, we simply `return` without doing anything else since there's nothing else we need to do. B-trees generally do not accept duplicate values.

However, if the `value` is *not* in the tree, the first item within `search_result` will be `None`. If this is the case, the second item of `search_result` will be the filename of the leaf node where the `value` *should* be inserted.

At this point, we call another method, `insert_into_node`, that places the `value` into this leaf node. This other method is essentially a continuation of the `insert` method, but I've moved the remaining logic into this separate method since we'll need to call on that same logic again in another context.

Let's dive into that `insert_into_node` method now. Here's the first chunk:

```
    def insert_into_node(self, node_file, value):
        node_data = self.read_node_file(node_file)
        values = node_data.get('values')
        children = node_data.get('children')
        parent = node_data.get('parent')
```

The `insert_into_node` method accepts the arguments `node_file` and `value` for the purpose of inserting the `value` into the `node_file`.

First, we call the helper method `read_node_file`, which reads the `node_file` and pulls out the node's `values`, `children`, and `parent`.

The code continues with:

```
values.append(value)
values.sort()
```

Here, we insert the `value` into the array of `values` that we pulled from the node. We then sort the `values` array to make sure that the new `value` ends up in the right spot.

At this point, our `values` array contains the modified node data, which now includes our inserted `value`. But we haven't yet modified the `node_file` itself. Before doing so, though, we need to first see if the current node is overstuffed. Hence, the next bit of code:

```
if len(values) > self.max_node_size:
    self.split_node(node_file, parent, values, children)
else:
    self.write_to_node_file(node_file, parent, values, children)
```

This conditional statement checks whether the current node has too many values. Let's first skip to the `else` clause, which occurs when we do *not* have too many values. In this simpler case, we call the `write_to_node_file` helper method, which overwrites the `node_file`'s values with the data from our new list of `values`.

If you take a quick glance at the `write_to_node_file` method, you'll see that it accepts *all* the details of a node, including its `parent` and `children` pointers. Here, though, we're passing in the `parent` and `children` that the node already has. The only thing we're overwriting in this context is the new list of `values`.

Now, let's go back to the first clause of the previous conditional statement. It handles a case where the node is now overstuffed. In this case, we call the `split_node` method, which we'll explore next.

I'll admit that the `split_node` method is a doozy, but I'll walk through it gently.

The `split_node` method signature goes like this:

```
def split_node(self, node_file, parent, values, children=None):
```

The method accepts a `node_file` representing the node we'll be splitting. Additionally, the method accepts the `parent`, `values`, and `children` of the node we'll be splitting.

In theory, once we are passing in the `node_file`, we shouldn't have to also pass in the `parent`, `values`, and `children` because that data can be read from the `node_file` itself. However, since at the time of calling `split_node` we already have that data handy, we may as well pass that data along and thereby avoid performing an extra I/O.

The `split_node` method kicks things off with the following line:

```
os.remove(node_file)
```

Let me explain. Our approach to splitting a node will be to create two brand-new node files and then delete the original node file. And so, this line uses Python's operating system module, called `os`, to delete the original `node_file`.

The next snippet creates filenames for the two new node files we'll be creating:

```
median_index = self.max_node_size // 2
left_node_filename = str(values[0]) + '.csv'
right_node_filename = str(values[median_index + 1]) + '.csv'
```

We split our node by creating a new “left” node and a new “right” node, each of which will contain half the contents of the original node. When choosing filenames for these new nodes, though, we need to be sure that we aren’t using a filename that already exists, since otherwise we’ll accidentally overwrite an existing node.

There are several ways we can go about this, but because this B-tree is designed to hold integers, I’ve decided to name the files after one of the integers within the node. Since we can be sure that a particular value will not exist more than once within the tree, we can also be sure that we’re not creating a filename for one node that already belongs to another node.

To do all this, the previous snippet first finds the `median_index`, which represents the index in `values` that begins the second half of the array. The `left_node_filename` is named after the first integer in `values`, while the `right_node_filename` is named after the item at `median_index`. Again, this is the first item in the second half of the array.

Next, we decide what node file will serve as the parent for our left and right nodes:

```
if parent == 'None':
    parent_node_filename = str(values[median_index]) + '.csv'
else:
    parent_node_filename = parent
```

That is, if the node we’re splitting doesn’t already have a parent, this means we have to create a brand-new node file to be the parent. (This will also become the new root of the tree.) We name the parent node file after the median value, which is going to live inside this node. If the node we’re splitting *does* have a parent, that existing parent will be the parent for our new left and right nodes.

In the snippet that follows, we deal with taking all the children pointers of the node we’re splitting and divvying them up between the new left and

right nodes. This is necessary only for nonleaf nodes, which are otherwise known as *internal* nodes. Internal nodes, by definition, contain children, so we need to split them up:

```
if children:
    left_children = children[:median_index + 1]
    right_children = children[median_index + 1:]
else:
    left_children = None
    right_children = None
```

If the node is an internal node, we split up the children equally, using the `median_index` as the halfway point. If the node doesn't have children, which is the case for leaf nodes, there are no children to split up.

The following code creates the left and right node files and fills them with the appropriate parent, values, and children pointers:

```
self.write_to_node_file(left_node_filename, parent_node_filename,
                        values[:median_index], left_children)
self.write_to_node_file(right_node_filename, parent_node_filename,
                        values[median_index + 1:], right_children)
```

Okay, we're making progress! We split up most of the current node's values into a left and right node, but there's still a median value that needs to be moved further up the tree.

The next snippet begins to deal with this:

```
if parent == 'None':
    new_parent = self.write_to_node_file(parent_node_filename,
      'None', [values[median_index]],
      [left_node_filename, right_node_filename])
```

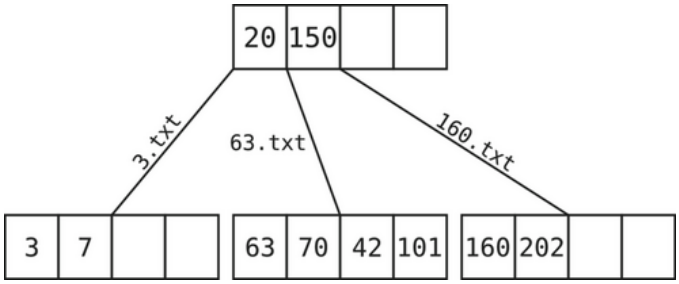
In this code, we state that if the split nodes don't have a parent, then we have to create a brand-new node. This new node will house the median value and become the parent of the split nodes. Additionally, this new node becomes the root of the tree.

However, if the split nodes *do* have a parent, we insert the median value into that parent. The next snippet handles this. But because the upcoming snippet is somewhat involved, let me first describe the high-level strategy here.

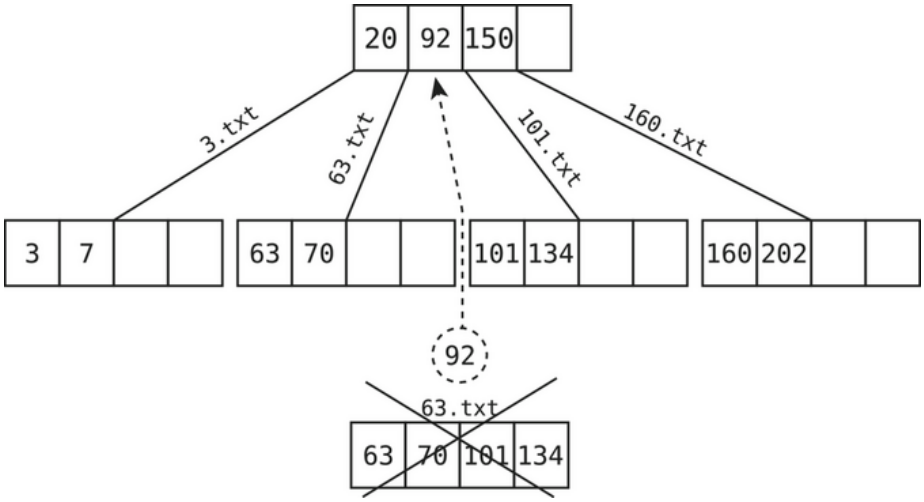
Wait—are you still here? You're *awesome*!

As I've said, we will insert the median value into the parent node. However, we *also* need to tell the parent node that it has two new children, namely, the recently created left and right nodes. On top of that, another thing we need to do to the parent is remove the pointer to the child node we deleted.

This is a somewhat delicate surgery. For example, say we have the following B-tree:



Note how the middle child pointer points to `63.txt`. If we now insert a 134, that `63.txt` node will split:



We insert the median value (92) into the parent, but also need to create two new child pointers. Now, just as the `63.txt` child pointer was in between `3.txt` and `160.txt`, we have to make sure that the new children pointers are *also* in between `3.txt` and `160.txt`.

The following code executes this strategy:

```
    else:
        parent_data = self.read_node_file(parent_node_filename)
        index_of_node_file = parent_data.get('children').index(node_file)

        updated_children = parent_data.get('children')[:index_of_node_file] + \
          [left_node_filename, right_node_filename] + \
          parent_data.get('children')[index_of_node_file + 1:]
        self.write_to_node_file(parent_node_filename,
          parent_data.get('parent'),
          parent_data.get('values'),
          updated_children)
```

We begin by reading the data from the parent node file. Then, we look at the parent node file's children and locate the precise index where it references the node we just split. Since we are deleting that node and replacing it with a left and right node, we need to update the parent node file accordingly to reflect this. That is, in the precise spot in the file where it references the split node's file, we remove the split node's filename and insert the filenames of the left and right nodes.

We accomplish this by setting an `updated_children` array. Here's what we do to create this variable:

1. We first include all the parent's children filenames up until, but *not* including, the filename of the current node.

2. We then add the filenames of the left and right nodes we've recently created.

3. Finally, we add all the remaining children filenames. These are the filenames that were to the right of the filename of the current node.

At the end of the day, the `updated_children` array holds the parent's original children pointers, except that we replace the pointer to the node we've split with pointers to the left and right nodes. Afterwards, we call the `write_to_node_file` method to overwrite the parent's file.

At this point, the parent node now holds pointers to the correct children. However, we have not yet inserted the median value into the parent. We do this next:

```
self.insert_into_node(parent_node_filename, values[median_index])
```

This recursively calls the `insert_into_node` method (that we walked through previously) to insert the median value into the parent. It's recursive because our current `split_node` method was itself called by the `insert_into_node` method.

Okay, we're nearing the end of the `split_node` method. There's one last major step, and that's to deal with splitting an internal node. Whenever we split an internal node, it's because we've previously split a leaf node, and we're trying to insert the leaf node's median value into an internal node that's already full.

Now, here's the problem. When we first split the leaf node into two new nodes, we couldn't tell these two nodes who their parent is. It might be tempting to simply tell them that their parent is the deleted node's parent, but here's the catch: we're about to split and delete the parent!

Because of this, we can only tell the children who their parents are after we've done all the splitting and have created nodes that are here to stay. We do this with the following code:

```
if left_children:
```

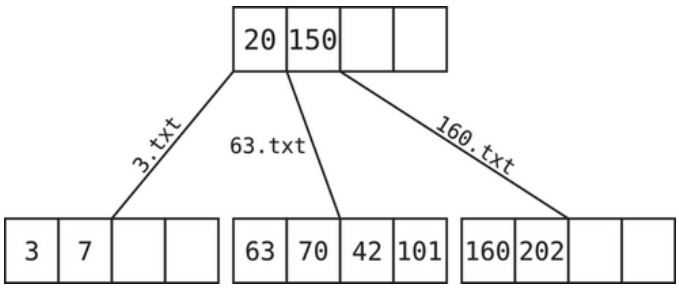
```
            for child in left_children:
                child_data = self.read_node_file(child)
                self.write_to_node_file(child, left_node_filename,
                  child_data.get('values'), child_data.get('children'))

        if right_children:
            for child in right_children:
                child_data = self.read_node_file(child)
                self.write_to_node_file(child, right_node_filename,
                  child_data.get('values'), child_data.get('children'))
```

That is, we tell the children of the left node that their parent is the `left_node_filename`, and we tell the right node's children that their parent is the `right_node_filename`.

AND. THAT'S. A. WRAP. Whew! We're done with B-tree insertion.

B-Tree Deletion

B-tree deletion follows yet another clever algorithm. Finding a value in a leaf node and deleting the value isn't a big deal itself, but what happens when we delete a value from an *internal* node? Let's take this tree, again, for an example:



Imagine that we have to delete the 20. We'd end up with a root node that has one value, but three children, which would completely mess up the B-tree structure.

Because of this, B-tree deletion does the opposite of what insertion does. While insertion causes nodes to split in order to grow the tree, deletion *fuses* empty (or close to empty) nodes together in order to *shrink* the tree.

More specifically, when deleting a value causes a node to be less than half full, that's when we start fusing nodes together. So if a node's maximum capacity is 10 values, fusing will potentially occur when a node ends up with only 4 values.

I will let you research the specific details of B-tree deletion if you're interested, as I need to save room in the book for some other important concepts.

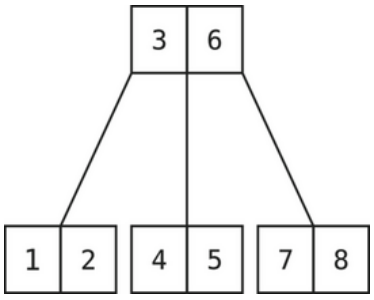
The Balance of B-Trees

Because B-trees grow (and shrink), they always maintain a kind of balance. To be specific, all leaf nodes of a B-tree are always on the same level of the tree. It's not like one path of the tree can end on the second level while another path stretches down to the fourth level. Because new values are only inserted into leaf nodes, and the tree only grows upwards, leaf nodes cannot possibly end up on different levels of the tree.

Another way to look at this is that if I try to make one branch of the tree fatter than the others, at some point, I'll end up failing. This is because if I make one node *too* fat, it ends up splitting! And once it splits, *all* parts of the tree grow at the same rate, and not only that branch.

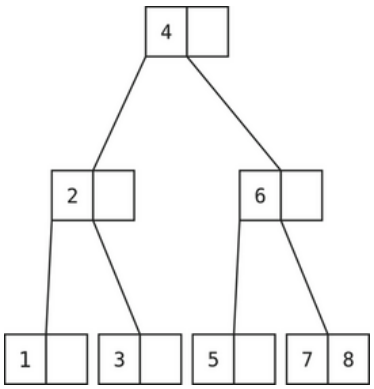
Ultimately, the fact that all leaf nodes live on the same level means that all paths from the root to any leaf node are all the same length. And so, in this sense, we can say that a B-tree always remains balanced.

That being said, we cannot say that a B-tree is always *perfectly* efficient. As with BSTs, inserting values in, say, perfectly ascending order will reduce the tree's efficiency to some extent. For example, look at the following B-tree:



This tree houses all integers from 1 through 8 on two levels efficiently, as all the nodes are full. Now, when building this tree, I happened to insert the integers in the sequence of 1-5-3-6-8-4-2-7, and it worked out great.

However, if I build the B-tree by inserting the same integers in perfectly ascending order, we get this tree:



You can see that this tree uses *three* levels while the previous tree used only *two*. Accordingly, we're not filling the nodes to capacity.

The reason this happens is that when we insert values in ascending order, we always insert each new value into the right-most leaf node. And so, while there are other leaf nodes to the left that have potential room to hold new values, we nonetheless ignore those leaf nodes and don't make use of their full capacity. Instead, we keep inserting into the right-most leaf node, which keeps causing it to split and grow the tree.

In other words, when inserting ascending values into a B-tree, we unnecessarily cause the B-tree to grow instead of first filling each node to capacity.

Ultimately, as with plain old BSTs, a B-tree could also benefit from randomizing the order in which we insert values.

B-Trees as Database Indexes

Let's now take a look at how B-trees may be used in the real world.

Many databases are organized as tables of columns and rows. An example database table is shown that stores data about houses for sale.

id	listing_date	address	city	state	style	asking_price
1	5/10/23	123 Main St.	Chicago	IL	Ranch	$567,500
2	8/20/23	456 Madison Ave.	New York	NY	Colonial	$1,055,555
3	9/4/22	789 Clark Rd.	Miami	FL	Tudor	$389,000
4	1/3/23	1011 Hudson St.	San Diego	CA	Art Deco	$440,000
5	6/16/21	1213 Xanthia Ave.	Milwaukee	WI	Victorian	$1,000,000
6	4/12/23	1415 State Blvd.	Salt Lake City	UT	Ranch	$875,000
7	10/30/22	1617 Broadway Ave.	Atlanta	GA	Ranch	$625,000

Only a small section of the table is shown, but you can see that it includes data regarding numerous house details, including the `address`, `style`, and `asking_price`.

Typically, databases use the `id` column to map each row to its location in memory. For example, if we ask the computer to find the row where the `id` is `7`, the computer can do so in one step since it knows the exact memory address where that row is located. In any case, in this table, the rows are sorted by the order of the `id`.

Now, let's imagine that our table contains *10 million* rows. If we were to ask the database to find a house located in, say, San Diego, it would have to perform a linear search on up to all 10 million rows to pick out a San Diego

home. The database cannot leverage the `id` in any way since the `id` has no correlation to the house's city. A San Diego house might belong to any `id`!

To combat this problem, many databases allow us to create something known as a *database index*. Here's an example of what a database index looks like conceptually:

city	id
Atlanta	7
Chicago	1
Miami	3
Milwaukee	5
New York	2
Salt Lake City	6
San Diego	4

This is a kind of mini-table that contains only the `city` and `id` columns of each house, and omits all the other attributes. And most importantly, this table keeps the rows in *alphabetical order by `city` name*.

(For simplicity's sake, this data only has one home in each city. Realistically, a single city such as `Atlanta` will have many homes. As such, the table would put all the `Atlanta`s next to each other, and our search will indeed return all those rows.)

Now, if this data structure were an ordered list, and we search for a San Diego home, we can now perform binary search on the list to find San Diego in speedy logarithmic time. And once the database finds the San

Diego row in the database index, it sees that the `id` is 4, and can, in one more step, retrieve the *entire* row of data for `id` 4.

Very often, we have to create *multiple* database indexes. This database index only helps us quickly find a home if we search for it by the `city` name. But if we plan on also searching for homes by, say, the `asking_price`, then we'd have to create a separate database index just for that.

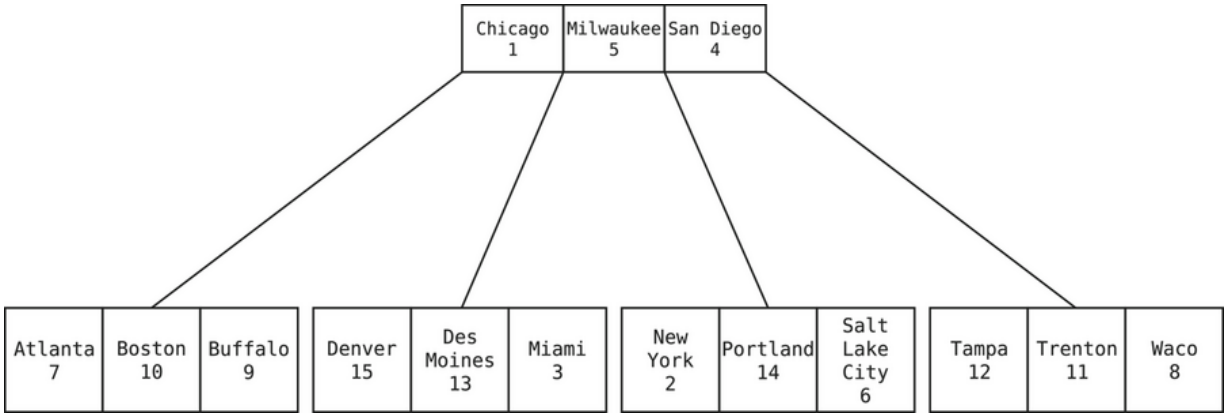
At the same time, we don't necessarily want to create database indexes for each and every column of a table, since database indexes involve a trade-off. That is, they can dramatically increase the speed of search, but they also can take up a lot of space. If our table contains 10 million rows, for example, then so too does a database index. This is a balance that database designers always need to consider.

There's another ramification of the fact that database indexes can consume a lot of space, namely, that often *the index is too large to fit into main memory*. Now, this isn't a death knell. We could still store the index as an ordered list and simply keep it in a file in the filesystem.

However, if we structure our database index as an ordered list, our only option to search through it would be with external-memory binary search, which we've seen takes $\log_2 N$ I/Os. We can do much better with B-trees.

You've learned that B-trees offer a much faster search, namely, that of $\log_{B+1} N$ I/Os. It is for this reason that numerous database engines use B-trees to store database indexes. It allows us to keep a large database index in the filesystem but still offer a blazing-fast search.

For a visual, how the previous database index may be stored in a B-tree with nodes that contain a maximum of 3 values is shown.



This B-tree stores both the city and id in each slot, and the keys are arranged alphabetically by city name. We can search this tree with just $\log_{B+1} N$ I/Os. Once we find the city we're looking for, we can then see its associated id. We then use that id to immediately find the memory location of the entire row of data corresponding to that home.

While some databases use classical B-trees to hold their database indexes, other databases use variants of the B-tree, such as the B+ tree, the B* tree, and others. Feel free to research those and explore all the delicious B-tree flavors out there.

Wrapping Up

This chapter opened up a whole new world. When dealing with data too large for main memory, we have to think about our algorithms in a completely different way. Specifically, instead of measuring speed by counting steps, we need to count I/Os.

Similarly, when we want to optimize external-memory algorithms, we need to focus on reducing I/Os. One way to do this is to pack blocks with as much useful information as we can. We also discovered that B-trees do this and are a wonderful data structure for storing data in the filesystem. As such, they're popular in databases as well as many other applications.

The topic of external-memory algorithms is a broad subject unto itself, and can easily fill its own book. I won't be doing that here, but I *will* devote one more chapter to the topic.

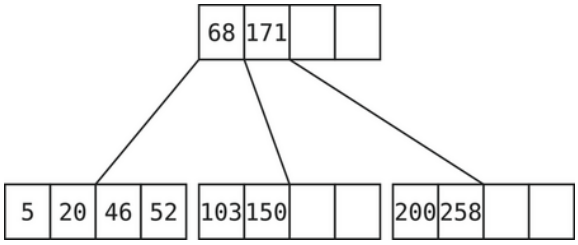
So, next up, we'll be discussing external-memory *sorting* algorithms. Onward we go!

Exercises

The following exercises provide you with the opportunity to practice with B-trees and external-memory algorithms. The solutions to these exercises are found in the section *Chapter 7*.

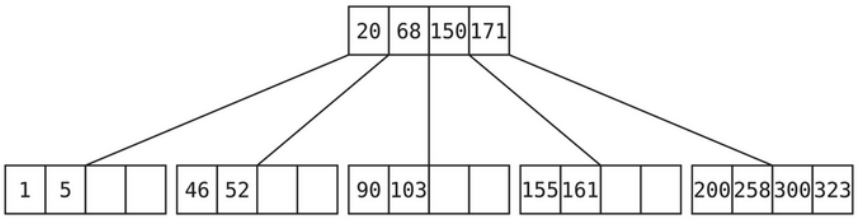
1. Let's say that we have a file in the filesystem that contains one million unsorted integers, and we want to find the greatest integer. The computer's block size is 500, and its RAM can hold a total of 10,000 values. How many I/Os will it take for us to find the greatest integer?

2. Take a look at this B-tree:



   What will the B-tree look like after inserting the following values in this order: 180, 85, 91, 117?

3. Here's another B-tree:



   What will the B-tree look like after inserting the following values in this order: 30, 40, 50?

4. Say that we have a B-tree containing 100,000 values, and each node can hold a maximum of 20 values. What is the greatest number of I/Os it would take to find any value?

Chapter 8

Wrangling Big Data with M/B-Way Mergesort

The previous chapter introduced the concept of external-memory algorithms and the unique approach for measuring and optimizing their time complexity. Most of the discussion revolved around optimizing external-memory *search* algorithms. But let's take this conversation to the next level and talk about how we might design external-memory *sorting* algorithms.

Consider the following scenario:

We want to sort a list of values. However, we have so much data that we can't fit all the values in RAM at once. How can we sort them?

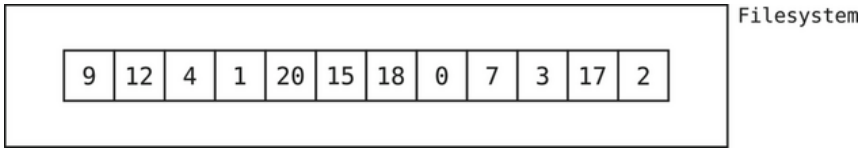
In this chapter, you'll learn the optimal solution for solving this problem. Along the way, you'll also learn about a useful *in-memory* algorithm for merging many lists simultaneously—a problem commonly known as *merging K sorted lists*. And most importantly, you'll gain further sophistication in analyzing external-memory algorithms and developing ways to optimize their speed.

External-Memory Sorting

In Volume 1, Chapter 5, I covered the good ol' simple sorting algorithm, Selection Sort. In a nutshell, Selection Sort performs a linear search through all the values in an array and locates the lowest value. Once you have the lowest value, you swap it with whatever value is at index 0. In the next round, you search all the values from index 1 and on, find the lowest one, and then swap the lowest value with the value at index 1. You then repeat this process for each subsequent index until you reach the end of the array. Selection Sort has a speed of $O(N^2)$.

But what would this process look like if the list of values is too large to fit into main memory all at once?

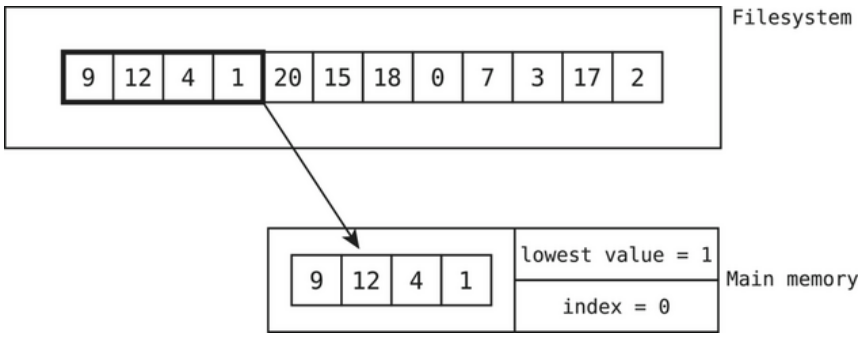
Here's an example of such a scenario. Suppose that our computer's block size is 4 integers. We'd like to sort the following list of integers that currently lives in the filesystem:



Before proceeding, I recommend that you try to work out an approach for sorting this data. You can try using a form of Selection Sort or any other sorting algorithm of your choice. I'll wait.

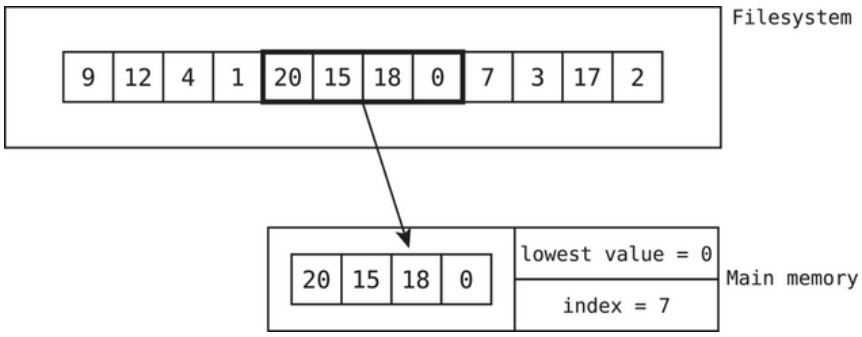
Okay, ready? Let's make an attempt at performing external-memory Selection Sort, and we'll count the number of I/Os along the way, as we discussed back in *Count I/Os, Not Steps*.

I/O #1: We load the first block into memory. We then do a linear search within this block to find the lowest value, which happens to be 1:



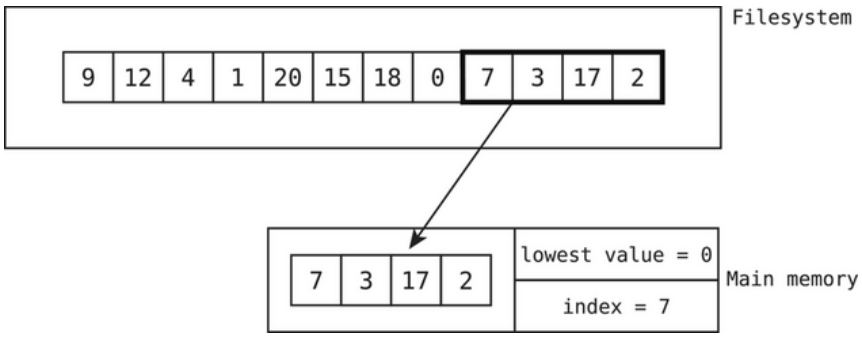
Note that our RAM technically needs to be able to hold *more* than 4 values since we have to hold the block *plus* a variable that keeps track of the lowest value. In truth, with Selection Sort we also need to keep track of the *index* of the lowest value. Let's assume that our RAM can hold at least 6 values, even though the block size is 4. This foreshadows an important idea we'll look at soon: a computer's main memory is generally larger than the block size. But let's keep things moving.

I/O #2: We load the second block and perform a linear search on it as shown in the first figure.



This block contains an even lower number than 1—it's a 0. The 0 is now our new lowest number.

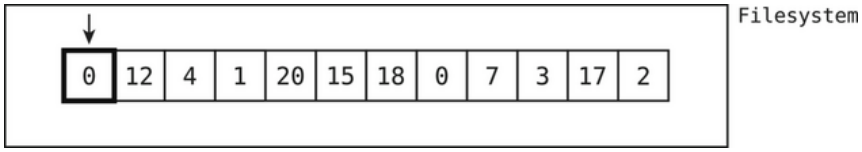
I/O #3: We load the third and final block as shown in the second figure.



This block's values are all greater than 0, so 0 remains the lowest value in the list.

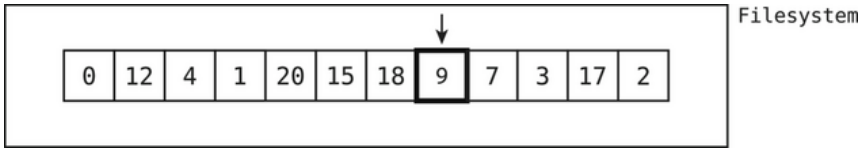
The next step of Selection Sort is to swap the 0 with the value that is at the beginning of the list, which happens to be the 9. However, this will take *two* additional I/Os.

I/O #4: We overwrite the first integer of the list and make it a 0.



This requires an I/O because the act of writing to a file interacts directly with the filesystem. Remember, the "I" of I/O stands for *input*, while the "O" stands for *output*. There's no better example of output than writing to a file. And so, this is a slow operation.

I/O #5: Similarly, we perform another I/O to overwrite the original 0 and replace it with a 9:



This took us a total of 5 I/Os. But lest you think we're done, we've only completed the *first round* of Selection Sort, as only the 0 is now in its proper place. We still have to sort all the other values!

The next three rounds (to sort the values at indexes 1, 2, and 3) will also each take 5 I/Os.

Technically, the next four rounds after that can use one fewer I/O since we don't need to load the first four values into memory anymore, as they'll already be sorted. We'd therefore be able to start the round by loading the block starting at index 4. As we progress through the rounds, the number of I/Os will decrease.

Big O of External-Memory Selection Sort

Let's now try to describe the speed of external-memory Selection Sort using Big O Notation. We'll start by analyzing the first round alone.

First, let's get our variables straight. We'll use N, as always, to represent the size of our data. In our example, this is 12 since our list contains 12 values. We also have the variable B, which represents the computer's block size. In our example, this is 4.

Now that our variables are in order, let's analyze how many I/Os our external-memory Selection Sort takes as it relates to N.

Because N is 12 and B is 4, we have to load 3 blocks to scan the entire list once, as 12 divided by 4 is 3. To put this in terms of N and B, we'd say that we need to perform N/B I/Os to scan the list once.

Now, we also need to perform two final I/Os to swap the values. This means that the first round takes N/B + 2 I/Os. Because Big O ignores constants, though, we reduce this to O(N/B) I/Os. However, keep in mind that this O(N/B) I/Os only describes the time complexity of the first round of Selection Sort. We still have to factor in all the remaining rounds as well.

Let's pretend for a moment that each and every round must perform N/B I/Os to scan the list, just like the first round. Now, with Selection Sort, we perform as many rounds as there are data elements. It comes out that for

each of the N data elements, we perform N/B I/Os. This comes out to be N * N/B I/Os, which we can express as N^2/B I/Os.

In the previous section, I pointed out that the number of I/Os decreases as we progress through the rounds. Based on this, we don't truly perform N^2/B I/Os. Technically, we can subtract a certain number of I/Os. However, I explained in Volume 1, Chapter 6 that when we have multiple orders of N, Big O only keeps the highest order. That is, if we have, say, N^2+N steps, we'd simply say that the time complexity is O(N^2). So in our case, even if we have N^2/B - 6N I/Os, we'd drop the "- 6N" and remain with just O(N^2/B). In Big O Notation, we'd say that the time complexity of external-memory Selection Sort is O(N^2/B) I/Os.

This is most certainly a slow algorithm, but that shouldn't come as a surprise. After all, Selection Sort, even when performed in memory, is a slow sorting algorithm. As such, we probably don't want to use Selection Sort as our external-memory sorting algorithm either.

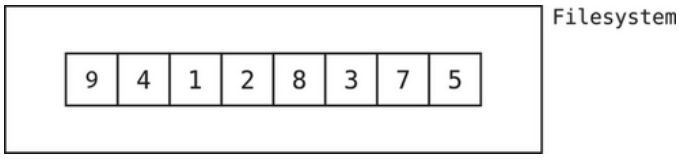
We've seen that there are much faster sorting algorithms out there, like Mergesort and Quicksort. Perhaps one of those could work well even when dealing with data in external memory. Let's try—you guessed it—Mergesort! And so, Mergesort comes back to haunt us once again.

A First Attempt: Two-Way External Mergesort

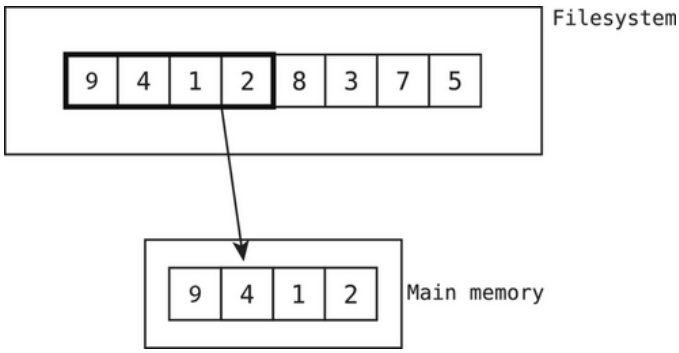
Over the course of this chapter, we'll make a few attempts at designing an effective Mergesort algorithm for external data. We'll start with a basic approach and optimize it for speed as we move along. By building up the concepts one layer at a time in this way, you'll have a much stronger appreciation and grasp of what will be our final algorithm.

Until this point, we've been using examples where the computer's memory blocks are the same size as RAM itself. In other words, RAM was just large enough to hold a single block. Indeed, we never cared about the RAM size in the first place. As long as it was large enough to hold a block, everything worked out nicely. However, having a computer that can only store one block in RAM will pose a significant hurdle for external-memory Mergesort.

In the following example, let's say that our block size is 4, and that main memory can only hold one block. Let's also say that we had to sort the following list of 8 integers from the filesystem:

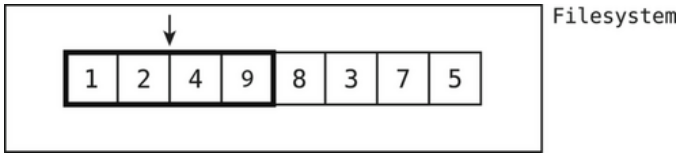


Okay, so the first thing we'd do is load our first block into memory:

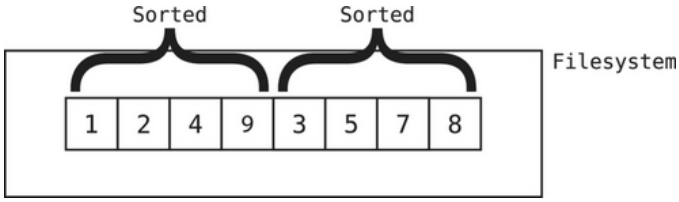


Once we have some data in memory, we can use *internal* Mergesort to sort the data within RAM.

With this data sorted, we can now spend another I/O to write these sorted values back to the filesystem:



Next, we'll do the same thing with the second half of the list. That is, we spend another two I/Os to load the second block and write those sorted values back to the file. And so, we now have a list whose two halves are sorted, but the entire list is not yet properly sorted:



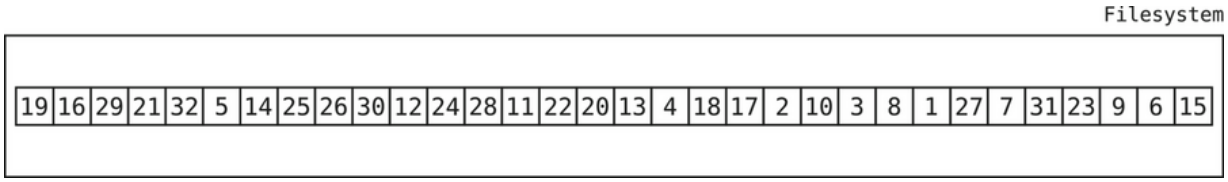
This is the perfect setup for Mergesort, as all we need to do is merge these two halves together.

But here's the catch. To merge the two halves, we have to be able to access *both* halves at the same time. However, each half is contained within one block, and if RAM can only hold one block at a time, we have no way to access both halves simultaneously! If our RAM can hold no more than a single block, we're kind of stuck.

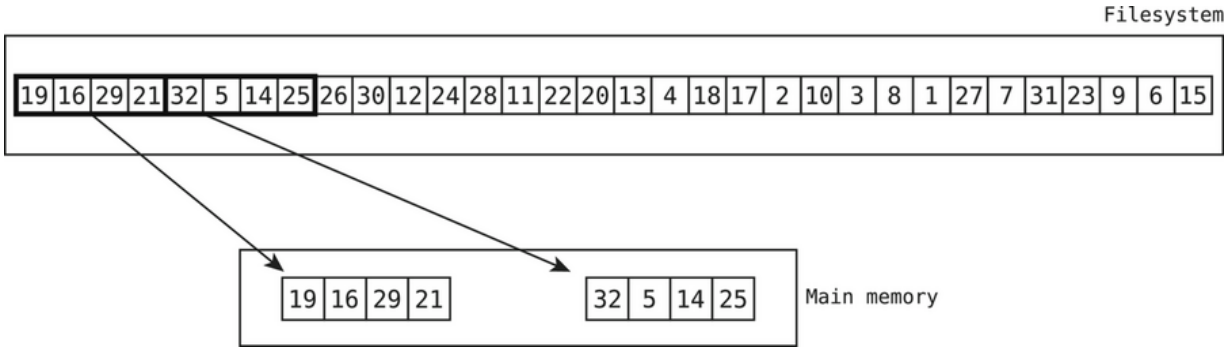
The good news is that in real life, a computer's RAM is much larger than the block size. And so, main memory has the capacity to hold *many* blocks.

In our example, we can solve everything by increasing our example computer's RAM to be able to store up to 8 values. This is equivalent to 2 blocks, which means that we can now proceed with external Mergesort.

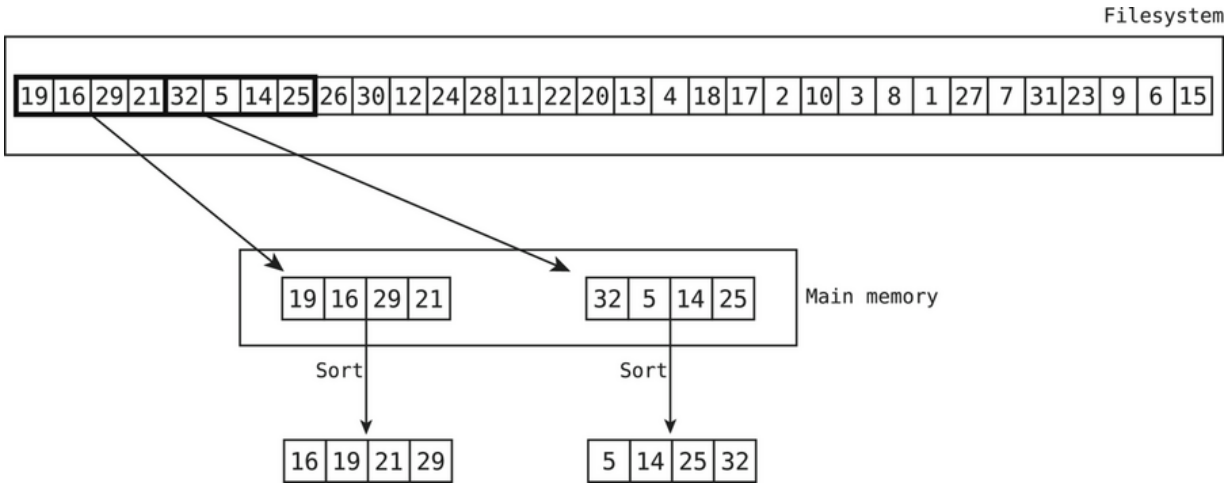
To make things fun, let's change up our scenario so that we'll be sorting 32 values. For example, here's a list of the integers from 1 through 32 in random order:



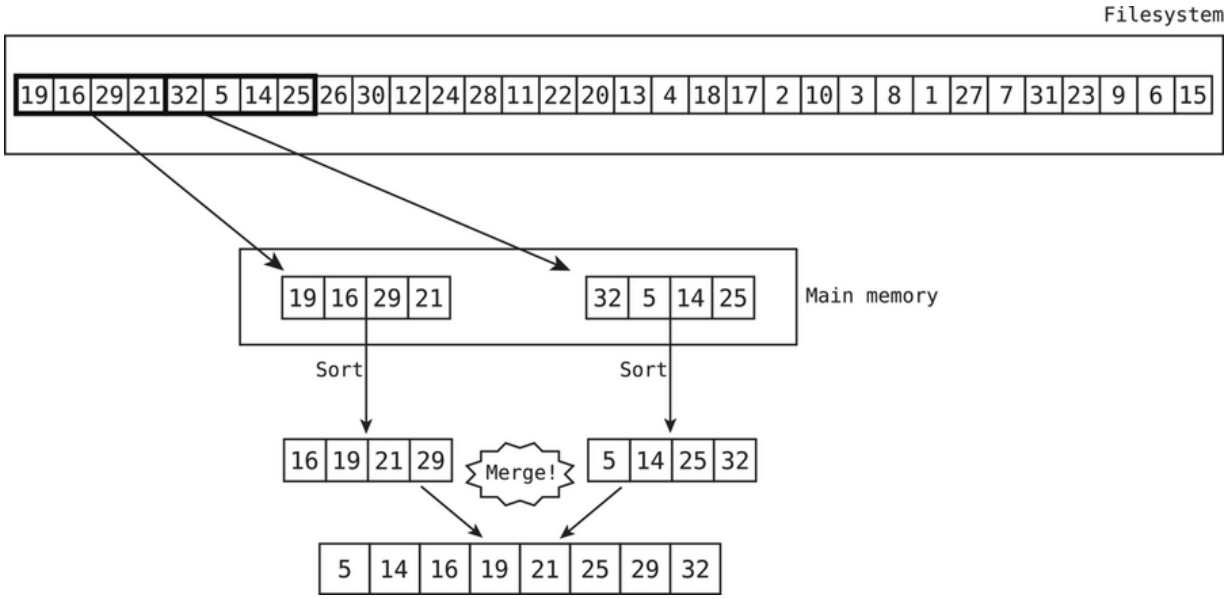
This list exists on the filesystem, and again, our goal is to sort the list. Let's start by loading the first two blocks into memory, which takes two I/Os:



Next up, we can sort each block independently. We can do this with internal Mergesort:



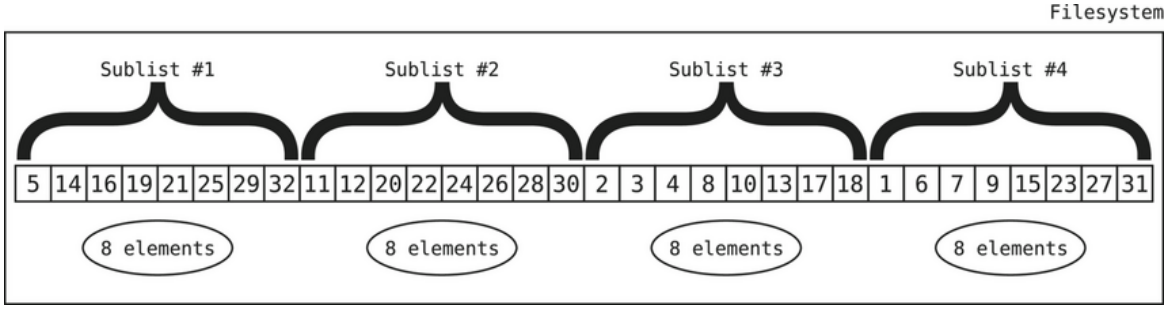
We then merge the two blocks:



Technically, this merge makes a copy of the original data, so our RAM might have to contain up to 4 blocks. Alternatively, when we merge the data, we can insert the data right back into the file instead of storing the sorted data in RAM.

In any case, we spend another two I/Os writing this sorted sublist back to disk.

Let's now skip ahead and say that we did this for the entire list, creating 4 ordered sublists, with each sublist containing 8 elements:



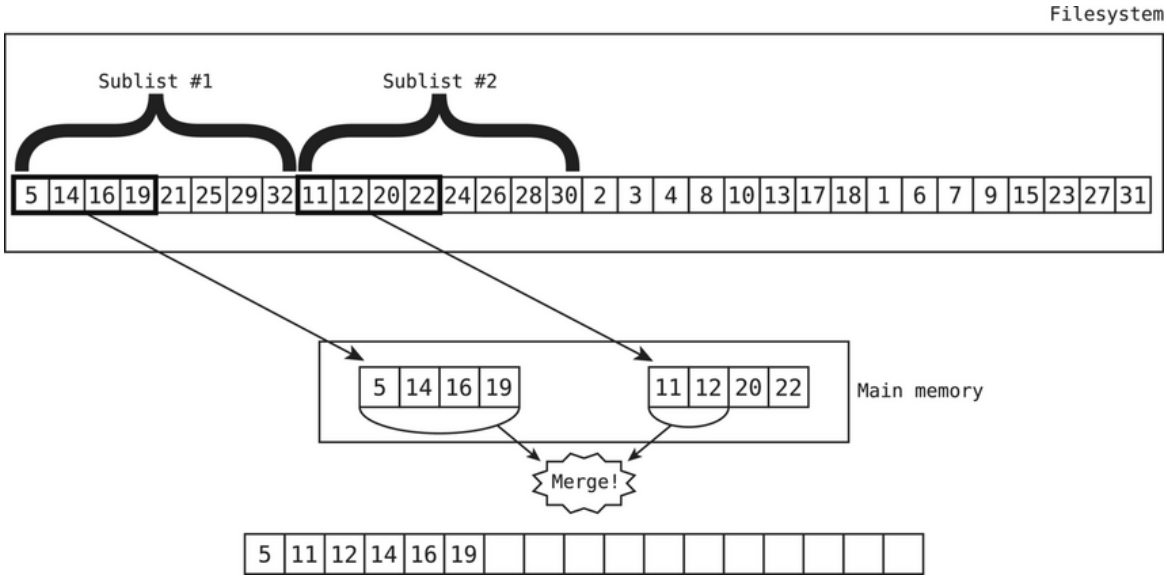
We now want to merge the sublists together to put the *entire* list in order. So our next immediate goal is to merge Sublist #1 with Sublist #2.

At first glance, though, it may seem impossible to merge sublists of 8 values if our RAM itself can only hold 8 values. How do we get *both*

sublists into memory?

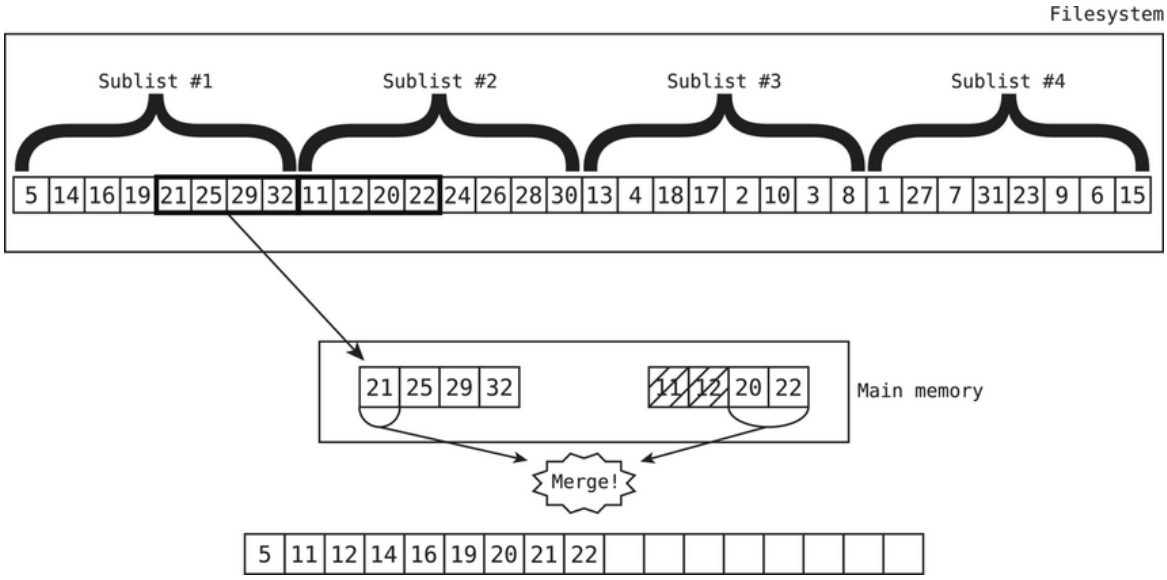
Well, *this* is how. Ready for a trick?

We load the first block of each sublist into memory. We begin by merging *most* of the values of the two blocks:

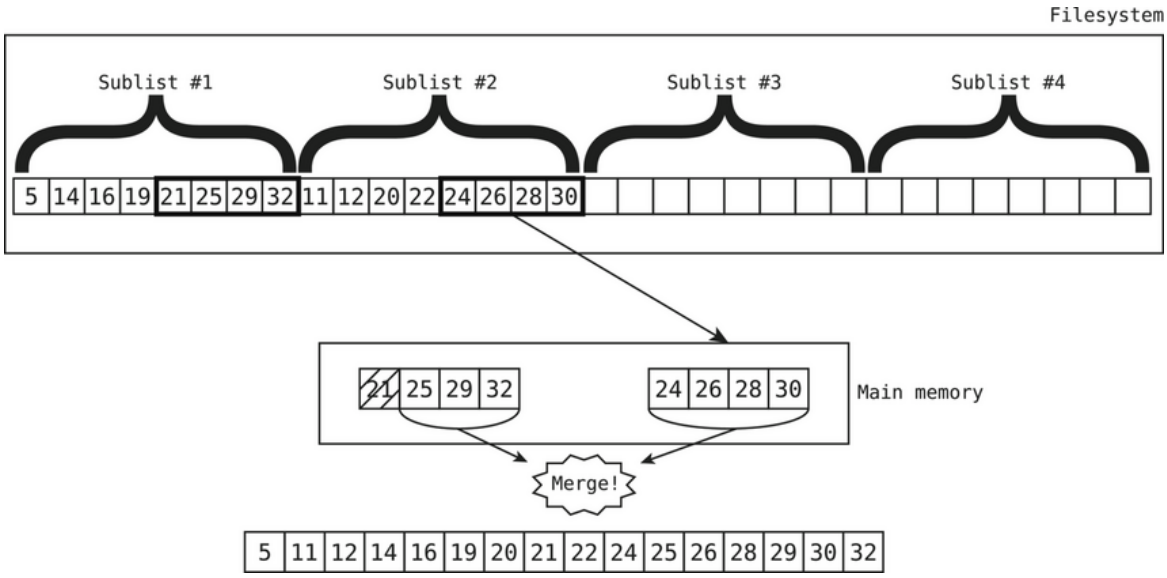


Here, we've merged the entire first block with the 11 and 12 from the second block, but we've paused there. This is because we cannot yet merge the 20 and 22 from the second block, since it's possible that the remainder of Sublist #1 still contains values that are less than 20 or 22. In fact, Sublist #1 contains a 21, and if we merged in the 20 and 22 now, the 21 would wind up being *after* the 22, which is just plain wrong.

What we do instead is that as soon as we complete merging values from one entire block, we load in the next block from the sublist that the completed block came from. In our example, because we've already merged all the values from one whole block, namely, the 5, 14, 16, and 19, and because this block came from Sublist #1, we now load the next block from Sublist #1:



In this image, we load in the second block from Sublist #1, which is the block containing the values 21, 25, 29, and 32. (I grayed out the 11 and 12 to indicate that they've already been merged previously.) We merge the 20 from the second block, the 21 from the first block, and then the 22 from the second block. At this point, we've exhausted the second block. Because this block came from Sublist #2, we now load in the next block from Sublist #2 and merge all the remaining values from Sublist #1 and Sublist #2:



We've now successfully merged Sublists #1 and #2, which is pretty exciting. But do keep in mind that this only represents half of our original list, which contains 4 sublists. As such, we then repeat these same steps for

Sublists #3 and #4. Once this is done, we are left with two sorted halves of our original list, with each half containing 16 elements.

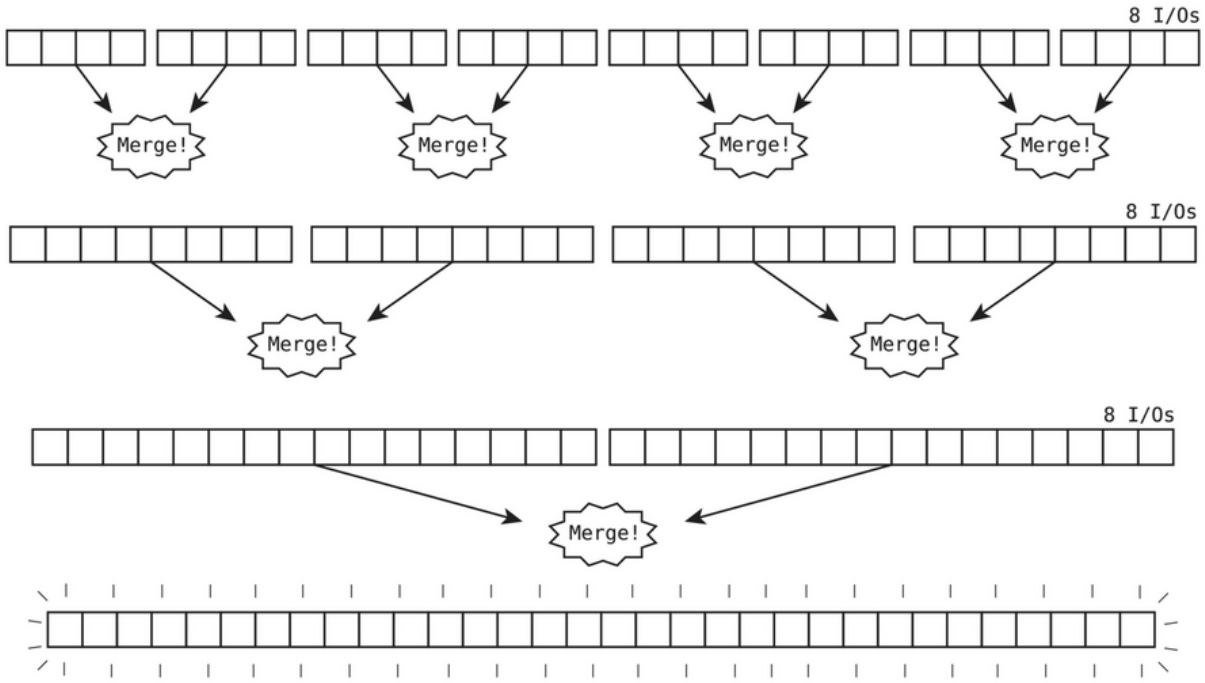
This brings us to the final phase of our algorithm, which is to merge the two 16-element lists together. Indeed, we can accomplish this the same way we merged the 8-element sublists together, which is to load one block from each 16-element list into memory and merge those blocks. As soon as we exhaust one block, we simply load the next block from that list.

It turns out that to execute this external-memory Mergesort algorithm, we require a RAM that can hold 2 blocks of data at once. Again, this is because we're continuously merging 2 sublists and using one block from each sublist to do so.

Because this algorithm's modus operandi is to keep merging 2 sublists at once, some call this algorithm *Two-Way External Mergesort*.

A Bird's-Eye View of Two-Way External Mergesort

Let's visualize Two-Way External Mergesort from a bird's-eye view:



The top row of this image depicts the initial merges that occur. To merge a block, we must load it into memory with an I/O, so it takes a total of 8 I/Os to conduct all the initial merges. (For now, we're not counting the I/Os of writing back to disk.)

Each of these merges produced a sublist of 8 elements, which can be seen on the second row from the top.

This second row shows how we then merged each pair of sublists together to get larger sublists of 16 elements. This takes another 8 I/Os, since we must load 8 blocks (of size 4) to load all 32 elements into RAM. This produces two larger sublists that each contain 16 elements.

The third row indicates that we spend yet another 8 I/Os to merge the two 16-element sublists. This finally produces the sorted list we've yearned for. At the end of the day, it took us a grand total of 24 I/Os of loading data into main memory.

It turns out that even with our computer's constraints of having a block size of 4 and a RAM size of 8, Two-Way External Mergesort is a fine algorithm

indeed. Had we used external-memory Selection Sort on the same 32 data elements, it would have taken us 256 I/Os!

However, with a little more RAM, we can design even *faster* versions of external-memory Mergesort.

M = Main Memory Size

We know that N represents the amount of data we're dealing with. In the prior example, N=32, since the list has 32 data elements. And we've also learned that B represents the computer's block size. In our example, B=4.

Now it's time to take a look at a new variable that is going to play an important role in our analysis of external-memory algorithms going forward: a variable that computer scientists call *M*. This M variable represents the *size of RAM*.

In the previous scenario, we used an example of RAM being able to contain 8 elements. And so, we'd say that M=8.

That's it; there's nothing earth-shattering here. It's just that going forward, I'm going to talk about M *a lot*. By digging into M further, you're going to level up your ability to analyze external-memory algorithms.

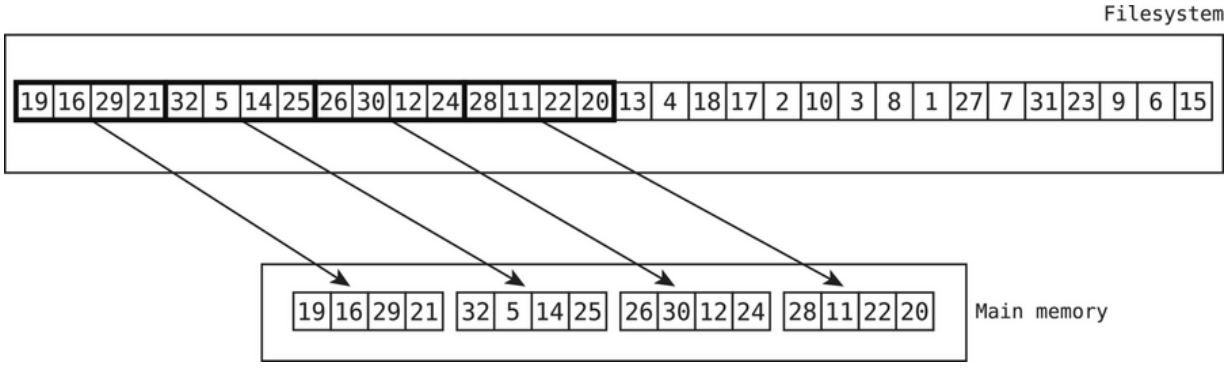
A Second Attempt at External Mergesort

Let's work with the same scenario of sorting the integers from 1 to 32. In this scenario, naturally, we'd say that N=32. And let's also continue using a computer whose block size is 4. In other words, B=4. But let's now upgrade our computer (for just $99!) so that its main memory has a capacity to hold *16* values. Using our new jargon, we'd say that M=16.

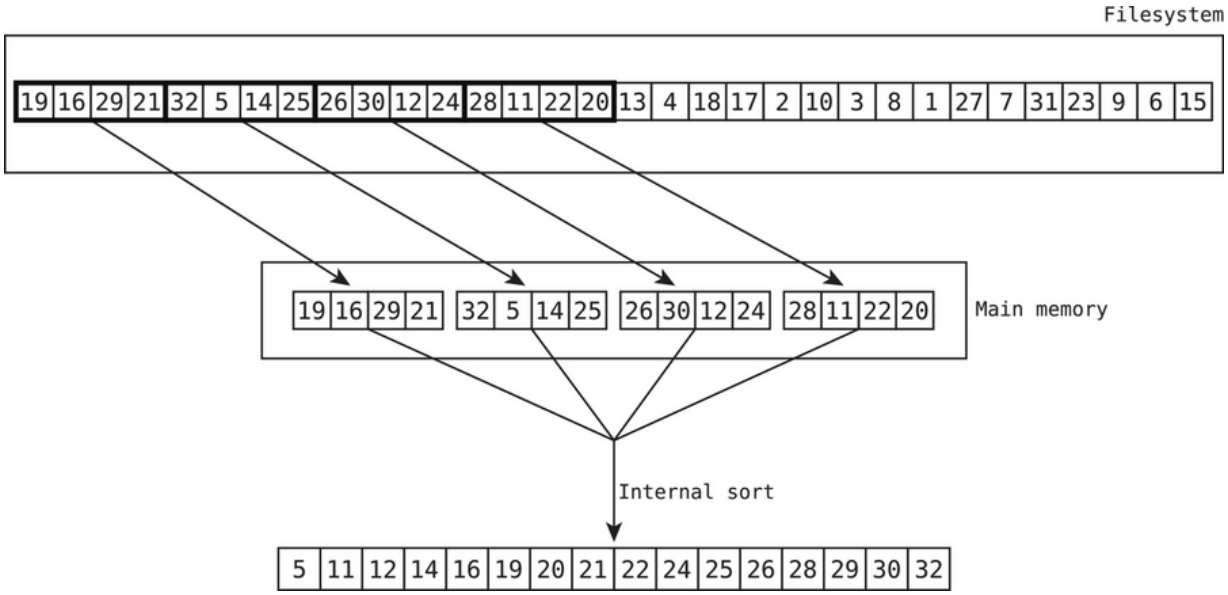
With this upgraded RAM, we can achieve external Mergesort using *only 16 I/Os* via what I'll call "Second-Attempt Mergesort." This algorithm will simply be a bridge between Two-Way External Mergesort and our final algorithm, so I won't even try to come up with a better name for it.

Second-Attempt Mergesort will take 16 I/Os to sort 32 values, which is faster than Two-Way External Mergesort, which took 24 I/Os. Second-Attempt Mergesort is similar to Two-Way External Mergesort, except that we can skip an entire set of steps.

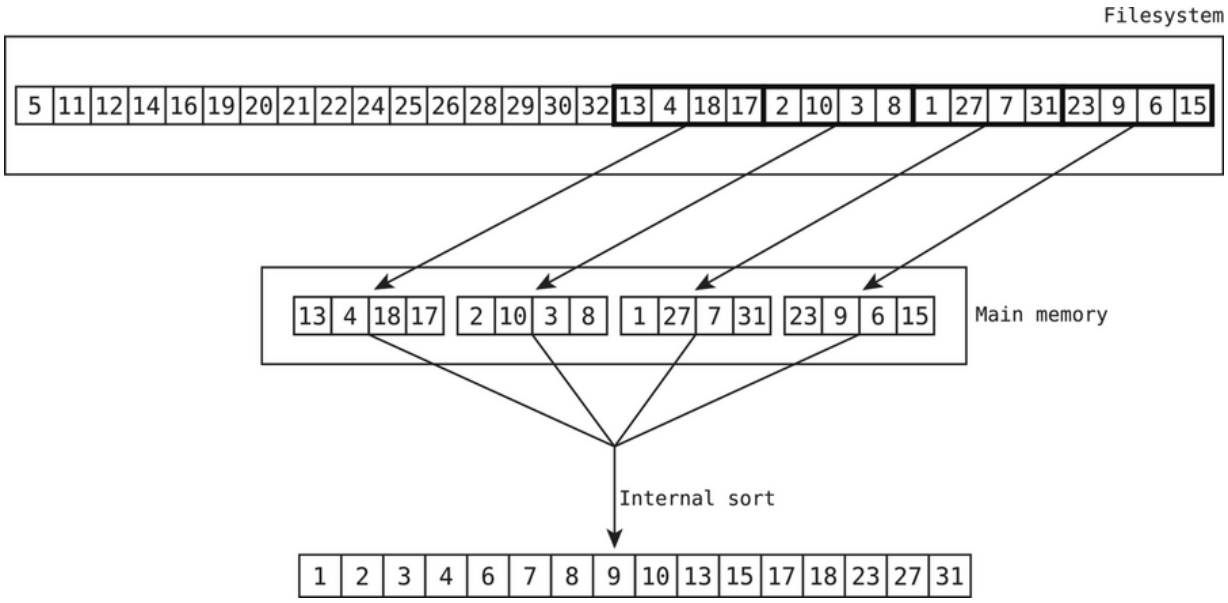
In Two-Way External Mergesort, we merged every pair of blocks to create sublists of size 8. However, since M=16, we can instead kick off our algorithm by filling up RAM with *as many blocks as possible*. With M=16 and B=4, this means we can load 4 blocks:



With 16 elements in RAM, we can use internal Mergesort (or any internal sorting algorithm, for that matter) to internally sort all 16 elements. And so, after 4 I/Os, we've already sorted half of the original list:



After writing this back to disk, we do the same with the second half of the list as shown in the [figure](#).

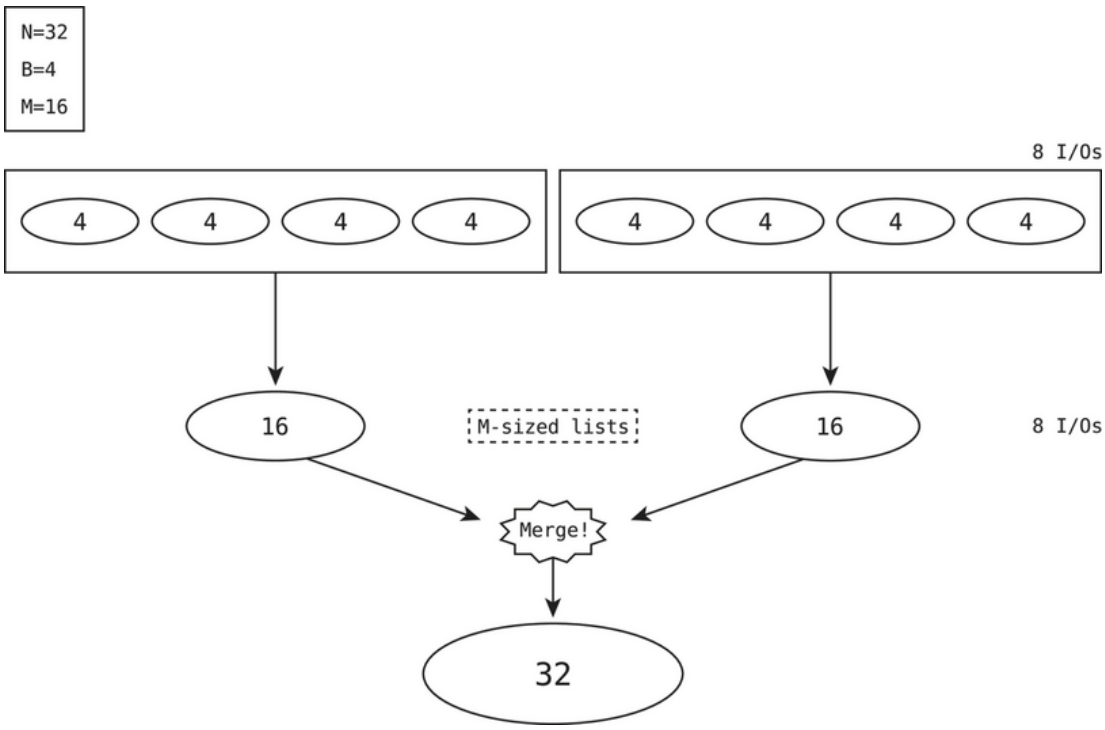


From here on, the rest of the algorithm is the same as Two-Way External Mergesort. That is, we perform another 8 I/Os to merge the list's two sorted halves together.

Essentially, what we accomplished here is that we skipped over the initial steps of merging 2 blocks at a time. There's no need to merge only 2 blocks

at a time if we can instead fill up RAM with 4 blocks at once and use an internal-memory algorithm to sort all the blocks' elements.

Let's use another bird's-eye view to visualize this new algorithm. Here, we're using a different, but simpler style of showing the data. Each oval represents a list or sublist of data with a particular size:



The top row shows that the original list is divided into 8 sublists of size 4, which is equal to our block size. These 8 sublists are divided into two rectangles that represent main memory, with each rectangle holding 4 sublists. This is because main memory, whose M=16, can hold a maximum of 4 sublists, which total 16 elements.

Loading these 8 sublists into RAM takes 8 I/Os since the job of an I/O is to load a single block into memory, and each sublist fits perfectly into one block. Each time RAM is filled with its maximum capacity of 4 sublists, it sorts the 4 sublists to create a single sorted sublist of 16 elements, which are depicted by the second row in the diagram. Note that I've labeled these lists

as being *M-sized lists* since each list indeed has a size that is equal to M. We'll talk more about the significance of the term "M-sized lists" soon.

The rest of the algorithm is identical to Two-Way External Mergesort. We take blocks from the two M-sized sublists and merge them together to create a final sorted list of size 32. This takes another 8 I/Os in total since we have to load all the elements again—which take up a total of 8 blocks—when doing the merging. This final merge produces a fully sorted list of size 32, which is seen on the bottom row.

In sum, the entire algorithm executes a grand total of 16 I/Os.

Before going on, I'd like to highlight a major takeaway from Second-Attempt Mergesort. First and foremost, we've discovered another general technique for optimizing external-memory algorithms. In the previous chapter, we looked at the trick of packing blocks with as much useful info as possible. Here, we discovered a new trick, which is to fill *RAM* with as much data as possible, if we can utilize an internal-memory algorithm to process that data. In our context, it's a waste to use an internal-memory sorting algorithm on only one or two blocks of data at a time. We may as well fill RAM to the brim, and sort all that data in memory all at once.

Don't Forget to Eat Your Variables

Let's now gear up to use Big O Notation to describe the speed of Second-Attempt Mergesort. Even though we'll ditch Second-Attempt Mergesort eventually, this same analysis will be used in computing the Big O of our final algorithm.

The first thing we need to do is figure out how to describe the number of I/Os that take place using our variables, N, M, and B.

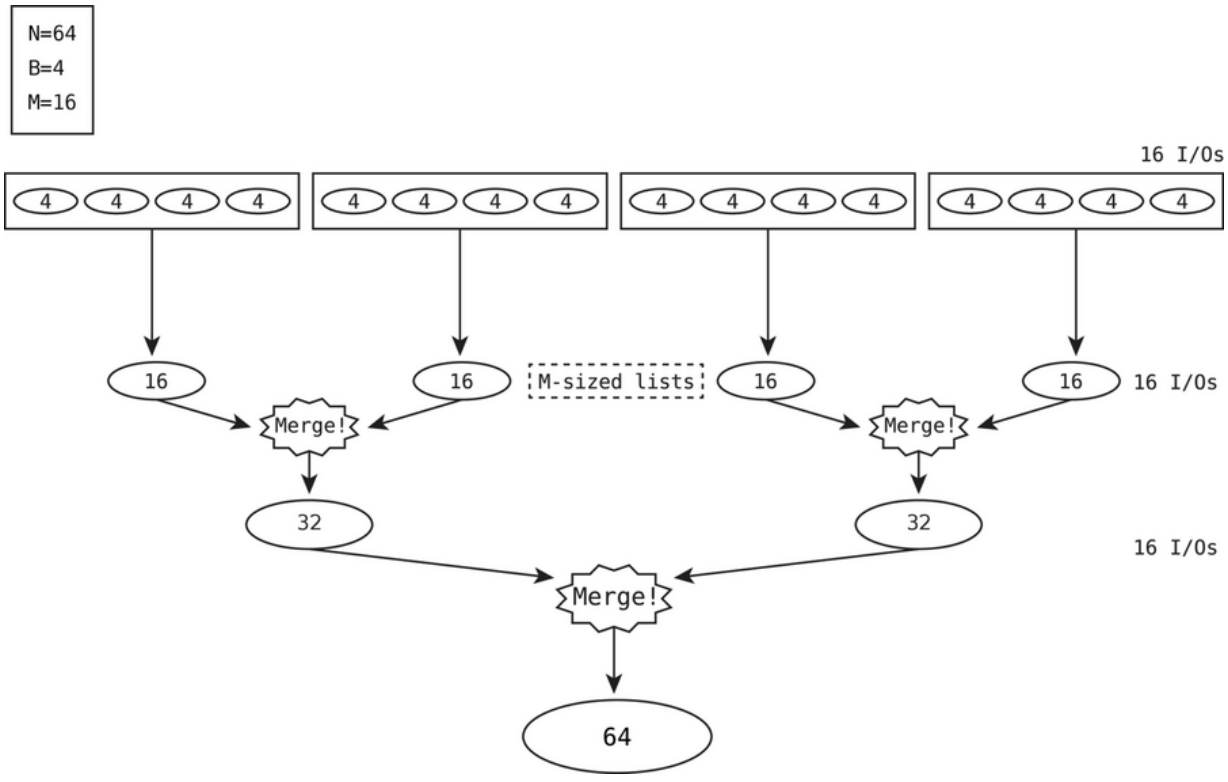
Recall that in the previous diagram, I highlighted in the second row that we created sorted sublists of size M. That is, because we filled RAM (whose size is M) completely before internally sorting the values, we ended up

producing *M-sized sorted sublists*. In our example, this meant that each sorted sublist had a size of 16.

With this in mind, we can use our shiny variables to describe how many M-size sublists we will be creating. In our example, N is 32 (there were 32 values), and M is 16 (our RAM can hold 16 values). Because 32/16=2, we created 2 sublists of size M.

We can now state this more generally: the first part of Second-Attempt Mergesort is to *create N/M lists* of size M. In other words, we used internal sorting to create a number of M-sized sublists. And how many M-sized sublists did we create? We created the same number of lists as whatever N/M computes to.

However, if we change up the scenario so that our initial list has *64* values (that is, N=64), this is what Second-Attempt Mergesort looks like now:



Once again, we begin by creating N/M sorted sublists of size M. In this case, this comes out to be 4 sorted sublists of size 16. We then merge each

pair of sublists to create 2 sublists of size 32. Finally, we merge those 2 sublists to create a fully sorted list of 64 values.

Big O of Second-Attempt Mergesort

Okay, we're almost ready to use Big O Notation to describe Second-Attempt Mergesort. This is a little more complex than our usual Big O analysis, but we can get there if we take one step at a time. Just tell yourself that it will be fun.

(If you're not having fun, that's okay; you can skip ahead. The Big O isn't critical to understanding the algorithms of this chapter. It just helps us contrast them all to each other in a quantitative way.)

In our most recent example scenario, we spent 16 I/Os in each phase of the algorithm. This is because each phase deals with 64 values, and it takes *16 blocks* (of size 4) to load these 64 values. And so, each phase took 16 I/Os. Now, the algorithm underwent *3 phases*, yielding a total of 48 I/Os since `16*3=48`. It would be reasonable to generalize this for all scenarios and say:

```
number of I/Os per phase * number of phases = total number of I/Os
```

I'll refer to this equation as our "Grand Formula." As you can see, we compute the total number of I/Os by multiplying two factors: the number of I/Os per phase and the number of phases. Since these factors are the key to our Grand Formula, I'll put them in quotes going forward to give them heightened importance.

So, the next thing we need to figure out is how to use our shiny variables to describe these "number of I/Os per phase" and the "number of phases." Then, we can plug those variables into the Grand Formula.

Calculating the Number of I/Os Per Phase

Let's start by using our variables to describe the "number of I/Os per phase." We know that in the previous example, there were 16 I/Os per

phase. This, again, was because each phase had to process 64 data elements, and when we divide these 64 values into blocks of size 4, we get 16 blocks. And as we've learned, it takes one I/O to load one block. Refer back to the image here to make sure this is clear.

As such, we get:

```
64 data elements / block size of 4 = 16 I/Os
```

With our variables, we can generalize this formula with:

```
N/B = the number of I/Os per phase
```

Tada! We've successfully used our shiny variables to calculate the "number of I/Os per phase." Let's plug what we have so far into the Grand Formula.

Again, let's take it from the top. Our Grand Formula to determine the Big O of Second-Attempt Mergesort was:

```
number of I/Os per phase * number of phases = total number of I/Os
```

Now, we've figured out that we can articulate the "number of I/Os per phase" as N/B. Accordingly, we can plug this into the Grand Formula, and come out with:

```
N/B * number of phases = total number of I/Os
```

Our next step is to determine how we can use our variables to articulate the "number of phases."

Calculating the Number of Phases

Look back again at the diagram here. Note that there are three phases. The first phase was spent creating the M-sized sublists, and the subsequent phases each cut the number of sublists in half. To make the following analysis easier, let's temporarily ignore the first phase and instead start our analysis from the second phase on.

The second phase starts with the M-sized sublists. To our great fortune, we've already determined earlier that there are N/M of these sublists.

Now, how many phases will it take to keep halving these N/M sublists until we get down to one single list? Well, that would be $\log_2$ N/M.

With that wrapped up, we can now return to the first phase.

We can treat the first phase as simply an extra phase that we add to $\log_2$ N/M. In other words, because the number of phases from the second phase and onward is $\log_2$ N/M, when we add the first phase, we get ($\log_2$ N/M) + 1. And because the "+ 1" is a constant, Big O ignores it, so we can articulate the number of phases as $\log_2$ N/M.

Now that we've calculated the "number of phases," we can plug that back into the Grand Formula.

As of this moment, here's what the Grand Formula currently looks like:

```
N/B * number of phases = total number of I/Os
```

Since we've determined that the "number of phases" is $\log_2$ N/M, we'll plug that in, leaving us with:

```
N/B * log2 N/M = total number of I/Os
```

We did it! We can finally conclude that in Big O, our Second-Attempt Mergesort algorithm has the time complexity of O(N/B * $\log_2$ N/M). In truth, Big O Notation drops the log base, but I'll leave it in because it'll allow us to make some important comparisons by the end of the chapter.

Now, I'll be the first to admit that when I look at the expression O(N/B * $\log_2$ N/M), I have trouble deriving much meaning from it. There are too many variables flying around. However, having this Big O benchmark will turn out to be useful when we contrast it with our final algorithm.

An Even Faster External-Memory Mergesort

I guess the heading spoils the surprise, but guess what? There's a *third* algorithm for external-memory Mergesort that is even *faster* than Second-Attempt Mergesort. In fact, this third algorithm isn't just faster; it's *way* faster. And it's the algorithm the pros use.

Prepare to have your mind blown.

But not yet. You see, we first need to explore a completely different algorithm. But don't fret. Although this may seem like another hurdle to jump over, this algorithm is super cool and worth learning about, even if you learned nothing else in this chapter.

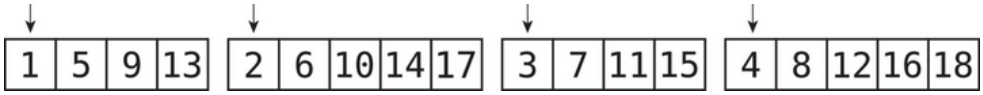
Merging K Sorted Lists

In this next section, we're going to temporarily leave the world of external-memory algorithms and return to some plain old *internal*-memory algorithms. Eventually, we'll tie this back into external-memory Mergesort. Okay, here goes.

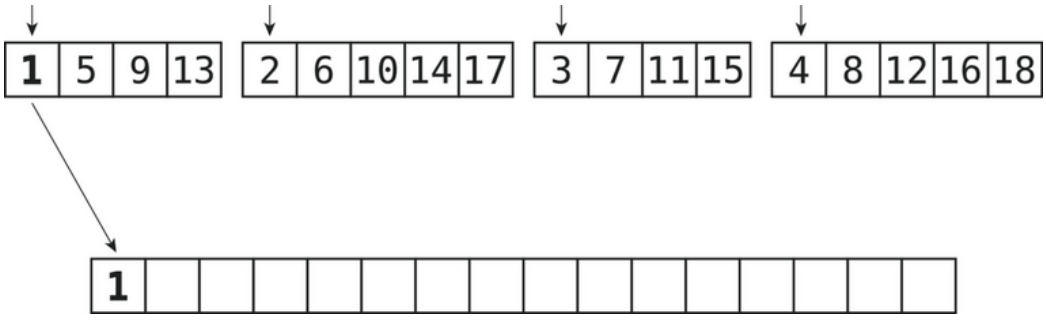
Whenever I've discussed merging lists throughout this book, I've always been referring to the idea of merging *two* lists at once. But let's say that we have *more* than two ordered lists, like three, or four, or seventy-six. How would we go about merging them? Again, right now I'm just talking about merging lists *in memory*. We're assuming that our RAM can comfortably accommodate all of our lists at the same time.

At first glance, we can apply the classical merging algorithm to as many lists as we'd like. For example, say we have four ordered lists:

Just as with merging two lists, we'd set pointers at the beginning of each list:

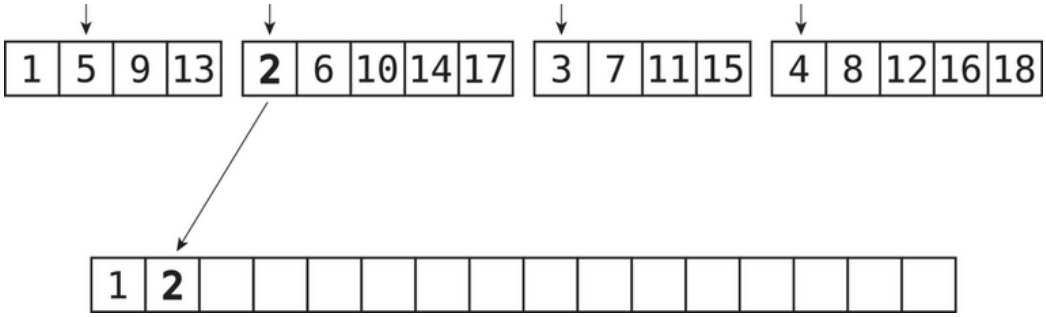


We would then compare all the pointers' values, find the lowest value, and insert it into what will eventually be the merged list. In this case, the 1 from the left-most list gets inserted:



Because we inserted a value from the left-most list, we move the pointer from that list one index to the right.

We then rinse and repeat. So again, we identify the lowest value from all the pointers. In this case, that would be the 2, so we insert it:



And so on and so forth. We repeat this process until all the lists have been merged. I'll refer to this algorithm as "naïve merging."

While this approach certainly gets the job done, let's analyze naïve merging's time complexity.

In this example, there are a total of 18 values. We spent 18 steps inserting each value into our final, sorted list. But in *addition* to insertion, we also performed comparisons. That is, before inserting any value, we had to look at 4 different pointer values to find the lowest value—one pointer from each list.

Now, looking at 4 different pointer values takes 4 steps. And we had to perform these 4 comparisons before each and every insertion. So, our total number of comparisons is:

```
18 values * 4 pointers = 72 total comparisons
```

When we add the 18 insertion steps to the 72 comparison steps, we have a grand total of 90 steps.

I also want to highlight a basic, but important idea: *the number of pointers is the same as the number of lists*. This is because we always maintain one

pointer per list.

With this in mind, we can also restate this equation as:

```
18 values * 4 lists = 72 total comparisons
```

Going forward, we'll talk in terms of the number of lists rather than the number of pointers; again, these numbers are one and the same.

Big O of Merging Multiple Lists

Let's now express all of this with Big O Notation.

We can say that N represents the total number of values across all the lists. In the previous example, N=18.

Now, we'll need another variable to represent the number of lists we're merging. For this, many computer scientists have decided to use K, which indeed is a nifty letter.

In fact, these same wonderful people also refer to our problem as *merging K sorted lists*. That is, instead of merging 2 sorted lists, we are merging *K* different sorted lists. In the previous example, K happened to be 4, but we can also have a scenario where we're merging 76 sorted lists, in which case K=76.

With our N and K variables in place, we can now express the total number of steps of naïve merging:

```
(N * K)      comparisons +
 N           insertions  =
____________________
(N * K) + N  steps
```

In Big O, we drop the "+N" since that's considered a lower order than N*K*. *And therefore, we'd say that naïve merging has a speed of *O(NK)*.*

Merging K Sorted Lists with Heaps

Now, there's a second approach we can use to merge K sorted lists. This approach involves the heap data structure, which I covered back in Volume 1, Chapter 16.

Here's a quick reminder of the points we need to recall about heaps for the sake of our discussion here:

1. Insertion into a heap has a speed of $\log_2 N$.

2. When we pop (in other words, remove) a value from a heap, we always get the lowest value from the heap.

3. Like insertion, popping also has a speed of $\log_2 N$.

In truth, a heap can arbitrarily be set up as either a *min-heap*, where popping from the heap gives us the lowest value, or as a *max-heap*, where we get the greatest value. For our purposes here, we'll use a min-heap, which is why I mentioned earlier that popping from a heap gives us the lowest value.

Now, how can we use a heap to merge multiple lists?

Perhaps the most straightforward thing to do is simply insert all the values from all the lists into the heap. We have N values to insert and each insertion takes $\log_2 N$ steps, so completing all the insertions will take $N \log_2 N$ time.

We can then pop all the values, one at a time, into our final list. Each pop gives us the lowest value currently in the heap. And so, as we keep popping values, we are effectively creating a perfectly sorted list. Indeed, this is a pretty famous algorithm called *Heapsort*.

Now, each pop takes $\log_2 N$ steps, and because we do it N times, we get a total of $N \log_2 N$ pops.

In sum, we have $N \log_2 N$ insertions and $N \log_2 N$ pops. While this is technically $2(N \log_2 N)$ steps, we drop the constant and have a sweet speed of $O(N \log_2 N)$. This algorithm doesn't care about the number of lists (K) because the same steps would occur no matter how many lists the N values are divided into.

So, is Heapsort's $O(N \log_2 N)$ better than naïve merging, which is O(NK)? Well, it depends on what K is. Say that N is 50 and K is 10. In this case, $O(N \log_2 N)$ comes out to be about 300. That is:

```
N  * log2 N =
50 * 6      =
300
```

By contrast, the O(NK) naïve merging algorithm would take 500 steps since:

```
N  * K =
50 * 10 =
500
```

In this scenario, where N=50 and K=10, the $O(N \log_2 N)$ Heapsort algorithm is faster.

However, in an alternative scenario where N is 50 but K is only *5*, we'd find that the O(NK) naïve merging algorithm is faster. In this case, the O(NK) algorithm will take only 250 steps since:

```
N  * K =
50 * 5 =
250
```

It emerges that the two algorithms are competitive, and which one is faster depends on how many lists there are. However, the top-grade approach I'll introduce next is faster than either of these algorithms in *all* cases.

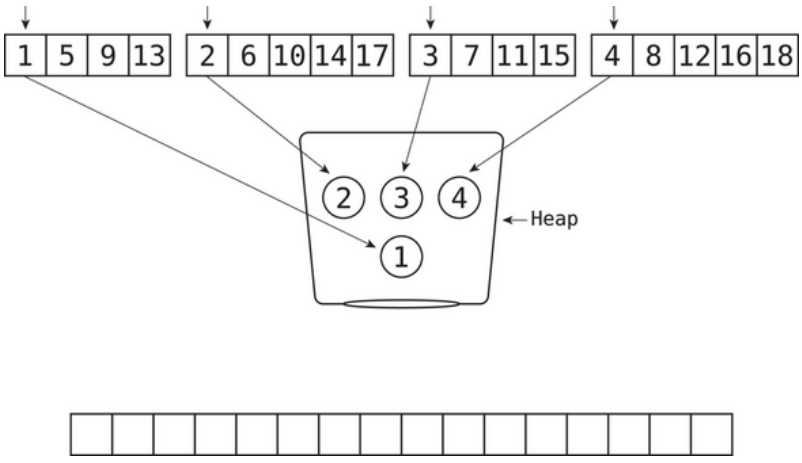
The Top-Grade Way to Merge K Sorted Lists

It turns out there's a *much* faster way of merging K sorted lists—faster than either approach we covered so far. This third algorithm doesn't have a special name that I'm aware of; it's simply the best way to merge K sorted lists. For the sake of clarity, though, I'll refer to this new algorithm as "Top-Grade Merge."

Now, the interesting thing is that Top-Grade Merge is a kind of hybrid of our previous two approaches! It uses a heap, but relies on the following epiphany: *we don't have to throw all the values into the heap at once*. The more the heap contains, the slower it is. All we truly need the heap to contain at a given time is *one value from each list*. But don't worry; this will all make more sense when we walk through the Top-Grade Merge algorithm, which we'll do now.

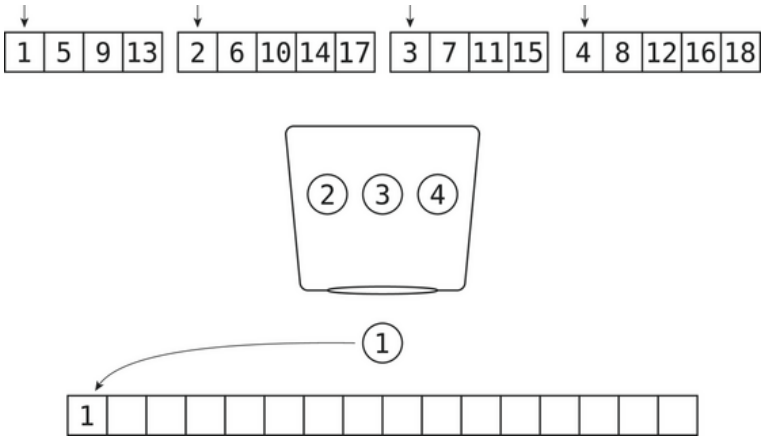
Here's how the Top-Grade Merge algorithm works:

Step 1: We start with pointers at the beginning of each list. We *also* insert all these values into our heap:

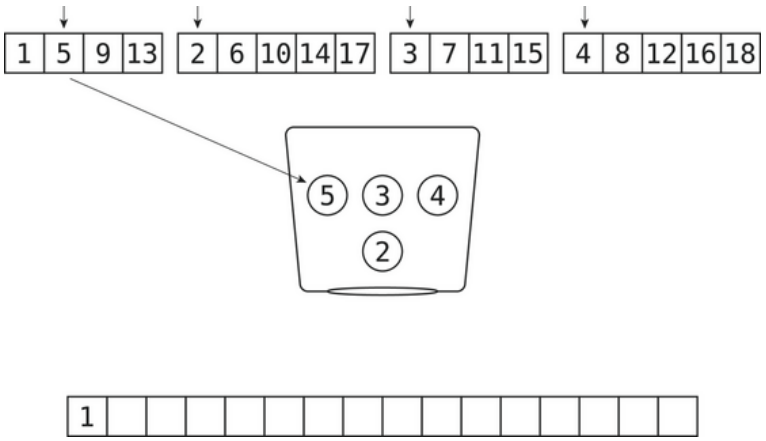


As you'll see, the heap will never contain more than 4 values. Also note that in these diagrams, I place the lowest value at the bottom of the heap.

Step 2: Next, we pop from the heap and insert the popped value into what will be our final sorted list. In this case, it's the 1 that gets popped:



Step 3: Now, here's the next major rule of the algorithm: we track down which list the popped value came from, and move that list's pointer to the right. We then place that pointer's value into the heap:



In this case, the popped value of 1 came from the left-most list, so we move the left-most list's pointer to the next index. This happens to be the 5, so we add the 5 to the heap.

From here on, we rinse and repeat. This means that next, we'd pop the 2 from the heap and insert it into the final list. Because the 2 came from the second list, we move the second list's pointer to the right. This points to the

6, which we then insert into the heap. We repeat this entire process until we've exhausted all the values from all the lists.

Here's the gist of why this algorithm is both effective and fast. It's similar to naïve merging, where we compared the lowest value from each list, picked out the lowest number, and inserted it into the final list. However, whereas in that approach we had to spend K steps comparing the K pointer values, in Top-Grade Merge, we rely on the *heap* to spit out the lowest of the K values. And the heap can do that much faster than K steps; it can do so in $log_2 K$ time.

Top-Grade Merge is also much faster than Heapsort, in which we simply threw all N values into the heap at once. When we do that, the heap's operations each take $O(\log_2 N)$ time. But in Top-Grade Merge, our heap needs only to contain K values at once. And K is smaller than N since the number of lists will certainly be smaller than the number of total values.

So, the heap's operations in Top-Grade Merge take $\log_2 K$ time, which can be considerably faster than $\log_2 N$ time. Therefore, Top-Grade Merge is faster than Heapsort, as the heap in Heapsort takes $\log_2 N$ time for each of its operations.

With all this in mind, let's calculate precisely how many steps take place in Top-Grade Merge.

Big O of Top-Grade Merge

Let's do the math. With Top-Grade Merge, we ultimately have to insert N values into the heap. We've also seen that heap insertion for Top-Grade Merge takes $\log_2 K$ steps. Because we spend $O(\log_2 K)$ time inserting each of the N values into the heap, this amounts to a total of $N * \log_2 K$ steps.

Now, we do also have to execute *another* $N * \log_2 K$ steps in *popping* the values from the heap. This gives us a grand total of $2(N * \log_2 K)$ steps. But

again, since Big O drops the constant of 2, we reduce this back to O(N $\log_2$ K).

And so, Top-Grade Merge has a total time complexity of O(N $\log_2$ K).

Comparing the Three Ways to Merge K Sorted Lists

We now have three different algorithms for merging K sorted lists with three different speeds. To sum it up:

- Naïve merging: O(NK)
- Heapsort: O(N $\log_2$ N)
- Top-Grade Merge: O(N $\log_2$ K)

We've already seen that sometimes naïve merging can be faster than heapsort, and in other scenarios, heapsort can be faster than naïve merging. However, Top-Grade Merge is *always* the fastest of the three. We've touched on the reasons for this a bit ago, but here's a crystal-clear summary.

When we compare Top-Grade Merge with naïve merging, it's pretty clear that Top-Grade Merge is the winner. This is because $\log_2$ K is always smaller than K itself. And so N * $\log_2$ K is definitely smaller than N*K.

Similarly, Top-Grade Merge is always faster than Heapsort since N * $\log_2$ K is smaller than N $\log_2$ N. This is because K is always smaller than N; the number of lists will certainly be smaller than the number of total values.

Let's take a look at how this all plays out in an example scenario.

Say that we have 10,000 values across 10 lists with 1,000 values in each list. In other words, N=10,000 and K=10. Here's how many steps each of the three algorithms would take:

- Naïve merging: 100,000 steps
- Heapsort: 140,000 steps

- Top-Grade Merge: 40,000 steps

As you can see, Top-Grade Merge is the fastest way to merge K sorted lists.

As I stated earlier, the entire discussion of merging K sorted lists has been in the context of merging data in main memory. However, we can now apply a similar idea to give a turbo boost to *external-memory Mergesort*. Remember that?

Code Implementation: Merge K Sorted Lists

I implemented a Python heap back in Volume 1, Chapter 16. The implementation here is basically the same, except that now we're using a min-heap instead of a max-heap. (For an explanation of the following code, refer to Volume 1, Chapter 16.) I've gone ahead and saved this in a file called `heap.py`:

```
class Heap:
    def __init__(self):
        self.data = []

    def root_node(self):
        return self.data[0]

    def last_node(self):
        return self.data[-1]

    def left_child_index(self, index):
        return (index * 2) + 1

    def right_child_index(self, index):
        return (index * 2) + 2

    def parent_index(self, index):
        return (index - 1) // 2

    def not_empty(self):
        return len(self.data) > 0
```

```
    def insert(self, value):
        self.data.append(value)
        new_node_index = len(self.data) - 1

        while (new_node_index > 0 and
               (self.data[new_node_index]
                < self.data[self.parent_index(new_node_index)])):

            parent_index = self.parent_index(new_node_index)
            self.data[parent_index], self.data[new_node_index] = \
                self.data[new_node_index], self.data[parent_index]

            new_node_index = parent_index

    def pop(self):
        if len(self.data) == 1:
            value_to_delete = self.data[0]
            self.data = []
            return value_to_delete

        value_to_delete = self.root_node()
        self.data[0] = self.data.pop()
        trickle_node_index = 0

        while self.has_smaller_child(trickle_node_index):
            smaller_child_index = \
                self.find_smaller_child_index(trickle_node_index)

            self.data[trickle_node_index], self.data[smaller_child_index] =
\
                self.data[smaller_child_index],
self.data[trickle_node_index]

            trickle_node_index = smaller_child_index

        return value_to_delete

    def has_smaller_child(self, index):
        return ((self.left_child_index(index) < len(self.data) and
                self.data[self.left_child_index(index)] < self.data[index])
                or
                (self.right_child_index(index) < len(self.data) and
                self.data[self.right_child_index(index)] <
```

```
self.data[index]))

    def find_smaller_child_index(self, index):
        if self.right_child_index(index) >= len(self.data):
            return self.left_child_index(index)

        if (self.data[self.right_child_index(index)]
                < self.data[self.left_child_index(index)]):
            return self.right_child_index(index)
        else:
            return self.left_child_index(index)
```

Armed with this min-heap, we can now implement the Top-Grade Merge algorithm for merging K sorted lists, as follows:

```
import heap as h

def merge_k_sorted_lists(lists):
    sorted_list = []
    pointers = []
    heap = h.Heap()

    for index, list in enumerate(lists):
        # We always insert into a heap an array that contains a value,
        # and an integer telling us which of the k sorted lists the value
        # came from. This integer is the index of the 'lists' array that
        # was inputted into this method.
        heap.insert([list[0], index])
        pointers.append(1)

    while heap.not_empty():
        popped_item = heap.pop()
        popped_value = popped_item[0]
        sorted_list.append(popped_value)

        # The current_list represents which list the popped value came from:
        current_list = popped_item[1]
        if pointers[current_list] < len(lists[current_list]):
            next_item_from_current_list = \
              lists[current_list][pointers[current_list]]
            heap.insert([next_item_from_current_list, current_list])
            pointers[current_list] += 1
```

```
    return sorted_list
```

Let's break this method down.

We first import the `heap` module so we can use it in our code. The `merge_k_sorted_lists` method begins like this, accepting an array of arrays called `lists`:

```
def merge_k_sorted_lists(lists):
    sorted_list = []
    pointers = []
    heap = h.Heap()
```

The purpose of this method is to take all the arrays within `lists` and return a single array containing all the values in ascending order. Here, we call that single array `sorted_list`, which starts out empty at the beginning.

We also set a `pointers` variable. This will keep track of all the pointers to the different lists. For example, say that we have four lists. Let's also say that the first list's pointer is currently at index 3, the second list's pointer is currently at index 5, the third list's pointer is currently at index 0, and the fourth list's pointer is currently at index 2. In this case, the `pointers` variable will hold the array `[3, 5, 0, 2]`.

Next, we create a heap, which we keep in a variable aptly named `heap`.

Our method continues by initiating a loop:

```
for index, list in enumerate(lists):
    heap.insert([list[0], index])
    pointers.append(1)
```

We iterate over all the arrays in `lists`. In this loop, we insert the first value from each `list` (that is, `list[0]`) into the heap. However, we don't insert the value alone. Instead, we wrap the value inside an array that *also* contains a

second item, namely, the identity of the `list` that the value came from. The identity of the `list` is represented by its `index` within the `lists` variable.

So, if we insert `[3, 1]` into the heap, that means we've inserted the value `3`, and indicated that this `3` originated from the *second* list. Again, the `1` represents the second list because `lists` contains all the original lists starting at index `0`, so `lists[1]` is the second list.

Because we've already inserted the first value from each list into the heap, we can start each list's pointer at `1` (the second index), so for each `list` we append a `1` into the `pointers` array.

Up until now, the method has largely been focused on setting things up. The remainder of the method represents the primary merging algorithm.

The merging algorithm is powered by a loop that runs as long as the heap contains anything. The reason for this is that, as I explained earlier, our merging is complete once the heap has been emptied completely. Here's the first part of the loop:

```
while heap.not_empty():
    popped_item = heap.pop()
    popped_value = popped_item[0]
    sorted_list.append(popped_value)
```

We pop the lowest item from the heap and put it in a variable called `popped_item`. As I explained earlier, this `popped_item` is not solely the lowest item. Rather, it's an array whose first value is the lowest item and whose second value is an integer pointing to the list where the lowest item originally came from.

We then append the `popped_value` to the `sorted_list`, which puts the `popped_value` in its proper sorted order.

The loop continues as follows:

```
    current_list = popped_item[1]
    if pointers[current_list] < len(lists[current_list]):
        next_item_from_current_list = lists[current_list]
    [pointers[current_list]]
        heap.insert([next_item_from_current_list, current_list])
        pointers[current_list] += 1
```

We use the integer `current_list` to point to the original list where the `popped_value` came from.

If the `current_list`'s pointer hasn't yet reached the end of that list, we grab the `next_item_from_current_list`, which is the value that the pointer of the `current_list` is pointing to. In other words, we're grabbing the next item from the `current_list`.

We then insert the `next_item_from_current_list` into the heap. Together with this value, we also insert the `current_list` so we can track which list this value came from. Before concluding our loop, we move the pointer from the `current_list` to the next index of that list.

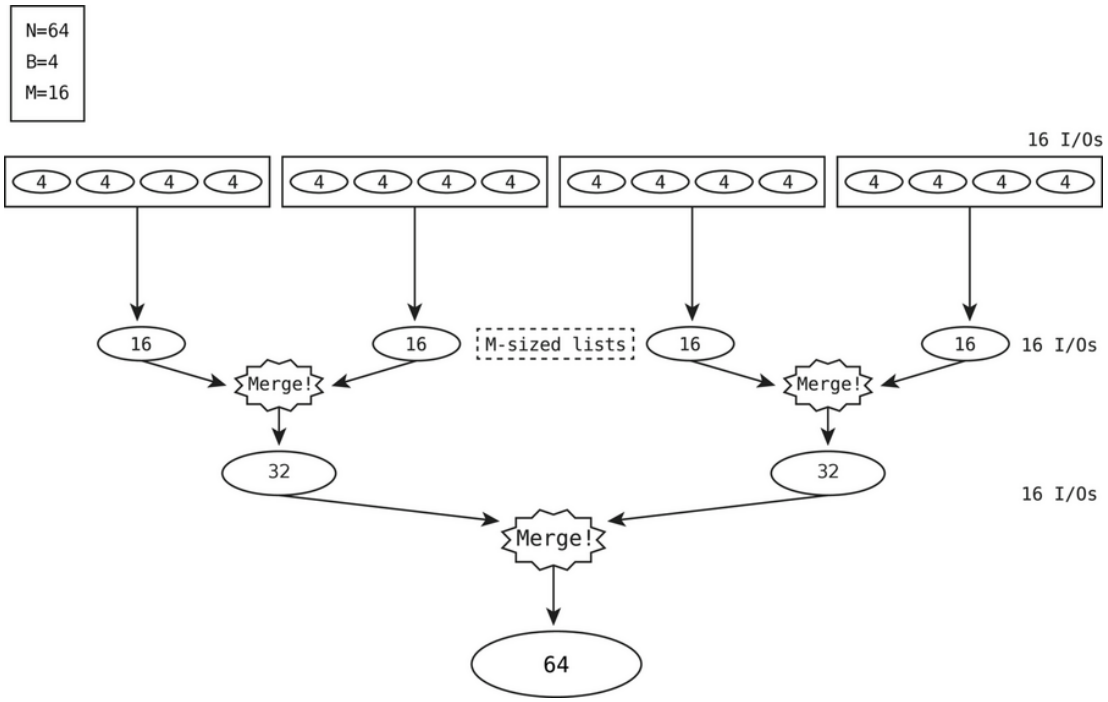
Once the loop is done, we return our `sorted_list`, which is the completely merged list.

M/B-Way Mergesort

You've seen many useful and interesting things in this chapter, and now we're going to put them all together. That's right—it's time to learn about the external-memory Mergesort algorithm of choice. It's called *M/B-Way Mergesort*, and you'll see soon where this algorithm gets its moniker.

Let's set the stage. We're back in a world where our data does *not* fit in RAM, so we need an external-memory approach for executing Mergesort.

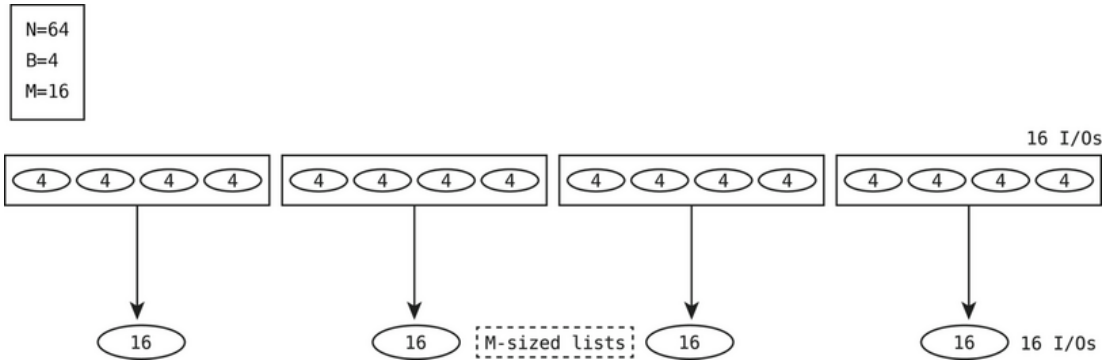
Say that we have a list of 64 data elements, and that our computer has a block size of 4 and a RAM size of 16. We've seen this scenario before, and our previous approach, which we called Second-Attempt Mergesort, worked like this:



We had three phases, each of which performed 16 I/Os. This yielded 48 I/Os in total. However, with M/B-Way Mergesort, we can accomplish Mergesort in *two* phases. Each phase will also take 16 I/Os, but because there are only two phases, we'll get a total of 32 I/Os.

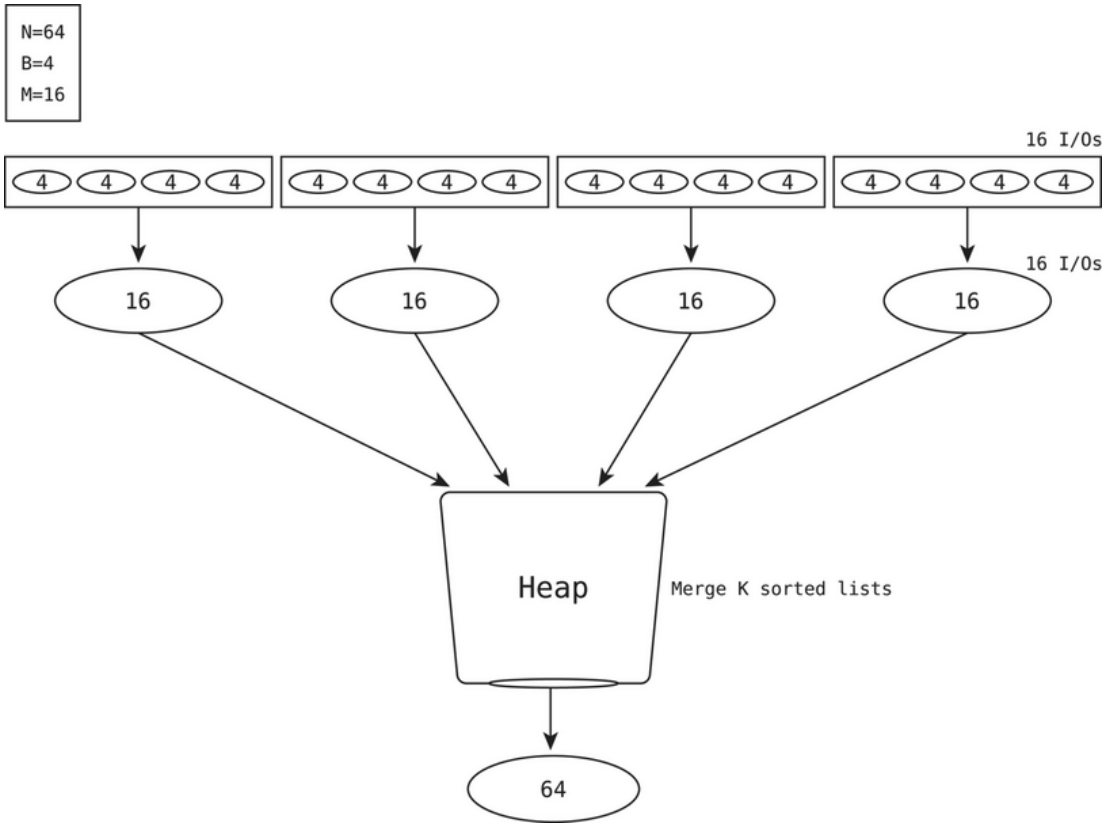
Here's how M/B-Way Mergesort works.

The first phase is identical to Second-Attempt Mergesort, in that we begin by creating 4 M-sized sublists:



However, instead of merging these M-sized lists two by two as we did in Second-Attempt Mergesort, we'll now do something different. Ready for it?

The next step is that we use the *Top-Grade Merge algorithm* to merge all 4 sublists into our final array. Here's an overview of what this looks like:



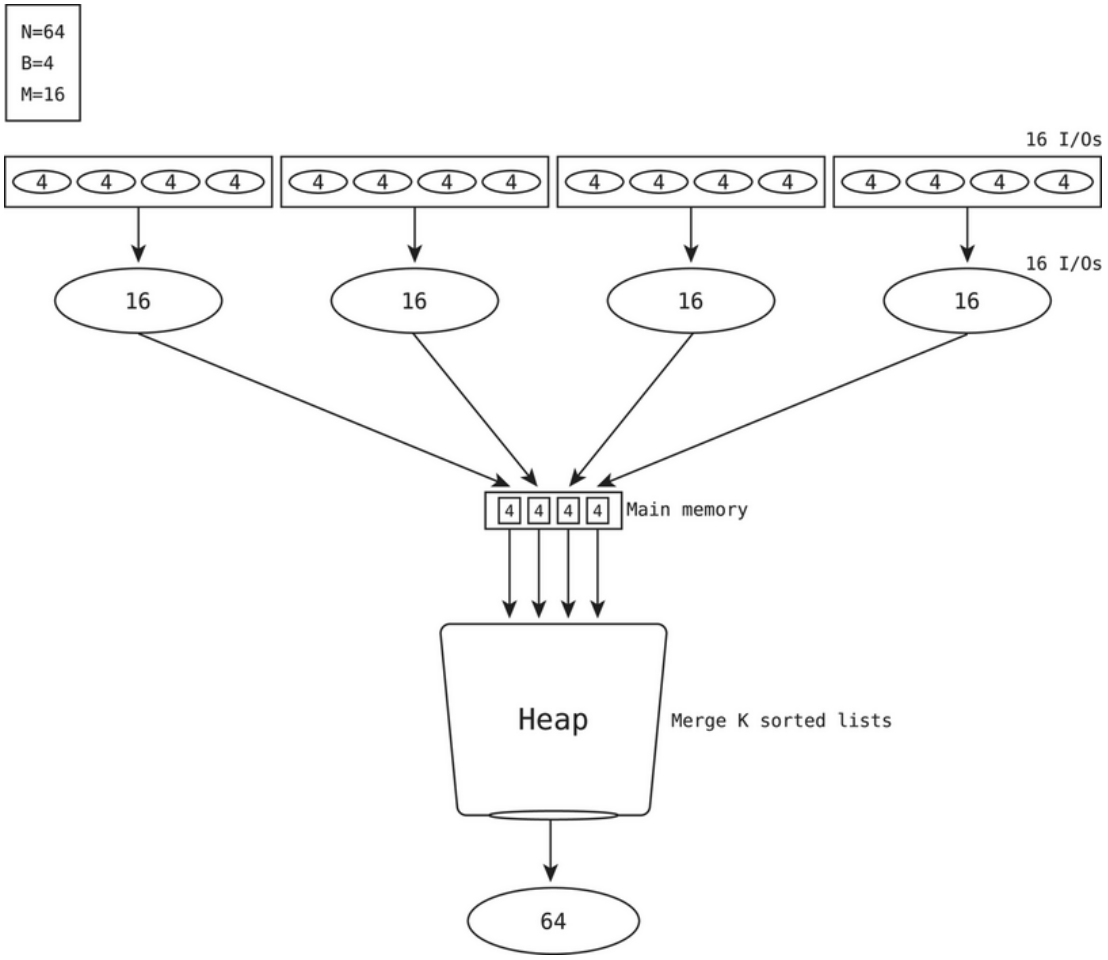
With this approach, we got the job done in just two phases! The first phase spent 16 I/Os loading all the elements and creating the M-sized lists. And the second phase spent another 16 I/Os moving all the elements into the heap. So, in two phases and 32 I/Os, M/B-Way Mergesort shaved off a lot of time.

With that overview taken care of, let's dig a bit further into the details.

In our context of external-memory sorting, we have to tweak Top-Grade Merge slightly. When I presented Top-Grade Merge earlier, it was as an internal-memory algorithm. In our scenario, though, there's a slight hurdle.

That is, we're trying to use Top-Grade Merge to merge 4 lists that each contain 16 values, which means that we're merging a total of 64 values at once. But if M=16, this means that RAM itself can only hold a max of 16 values at once. So, how can we merge 64 values at the same time?

The answer, though, is that we don't need to have all 64 values in RAM at once to perform Top-Grade Merge. Instead, we can use a technique that may be familiar to you now—we simply load one block from each list into RAM at once:



Here, we start by loading one block (of size 4) from each of the 16-element M-sized sublists into main memory. We then perform internal Top-Grade Merge on those four blocks, which are essentially four lists. As we exhaust each block, we pull the next block from its associated 16-element sublist. Note that when this diagram shows the four blocks in main memory, it's a snapshot in time. Eventually, all 64 elements from our original list have to pass through main memory. However, at a given moment in time, there are up to 4 blocks (totaling 16 elements) in main memory.

Just to be super clear: when the diagram shows the 4 blocks of 16 values being inserted into the heap, we're not inserting all 16 values at once. Again, we're performing Top-Grade Merge, which means that we're only inserting *one* value from each block into the heap, and the heap will hold no more than 4 values at once. The diagram shows the process from a high level.

Now, the heap has to live in main memory, too. In truth, we need our RAM to be large enough to accommodate the blocks *and* the heap. The diagram doesn't depict the fact that the heap also lives in main memory, but keep in mind that this will need to be the case.

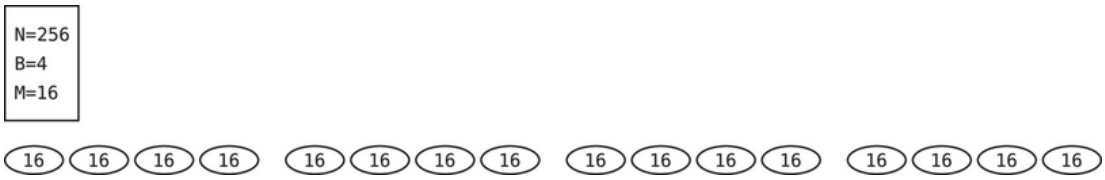
In short, we merge K sorted lists by merging K sorted *blocks*. Once we've exhausted a particular block, we load the next block from the list where that block came from. At the end of the day, the result is the same as merging K sorted lists.

Now, while this is all very awesome, there's one last hiccup to deal with.

What's M/B All About?

At first glance, it may seem that M/B-Way Mergesort will *always* get the job done in two phases. However, this isn't the case.

To demonstrate, let's change up our scenario so that N=256. Assuming that the other variables remain the same (M=16 and B=4), our first phase will produce *16* sorted M-sized sublists:



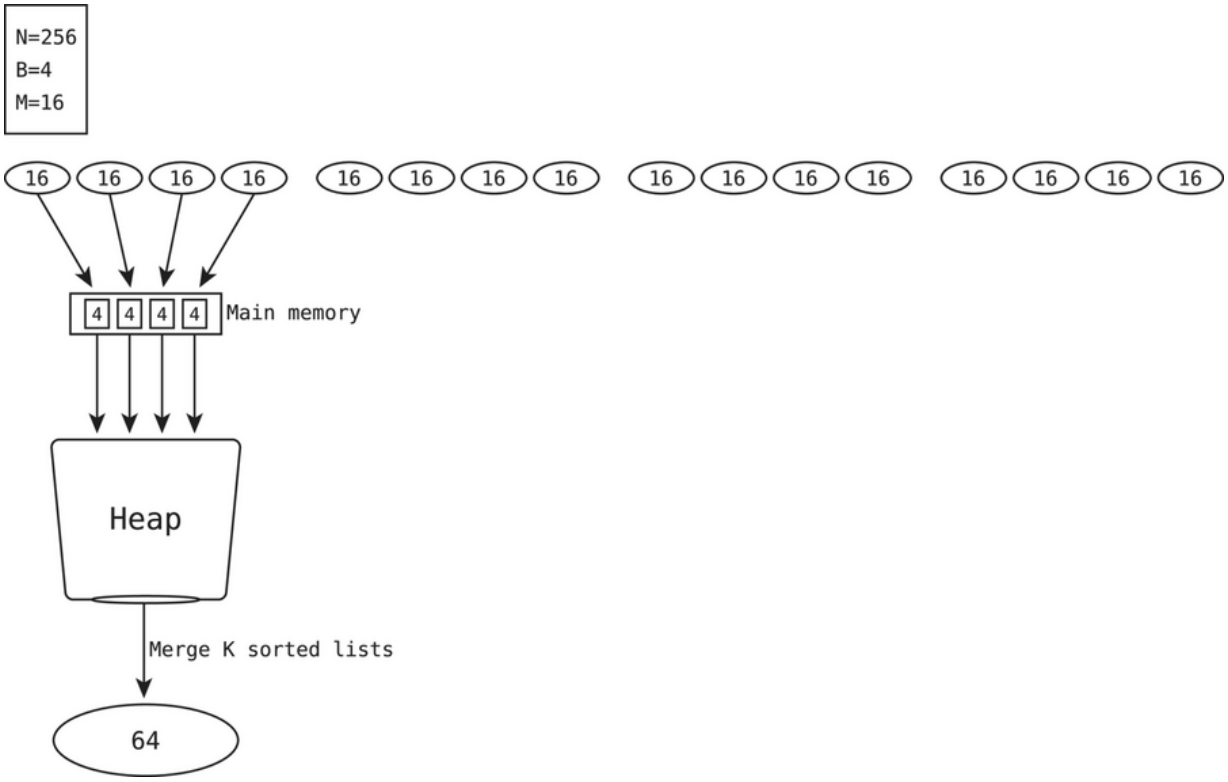
These M-sized sublists are now back living in external memory once again.

For the next phase, we'd attempt to use Top-Grade Merge just as we've done before, but there's a little catch.

As I've explained in the previous section, to perform Top-Grade Merge, we need to load one block from each and every sublist. And this is where we hit a wall. That is, in our scenario, K is 16, meaning that we have 16 sublists. As such, this would require loading 16 blocks. But because B=4, this means that loading 16 blocks means loading 64 values in total. But if

M=16, we don't have room for all 16 blocks; we only have room for 16 *elements*! In our scenario, *RAM can only hold a maximum of 4 blocks.*

So, here's what we do. We don't perform Top-Grade Merge on all 16 sublists at once. Instead, we merge 4 sublists at a time:



The reason why we choose to merge specifically *4* sublists at once is because RAM can hold a max of *4* blocks at once. Here, we load one block from each of the sublists into main memory. This fits perfectly into our main memory of size 16, as each of the 4 blocks contains 4 data elements, totaling 16 elements.

We then perform the same algorithm as before, which gives us a sorted sublist of 64 elements. This is the oval at the bottom of the diagram.

We're not done with the algorithm yet, but let me highlight an important takeaway: *the greatest number of sublists we can merge at once is identical to the maximum number of blocks that RAM can hold at once*.

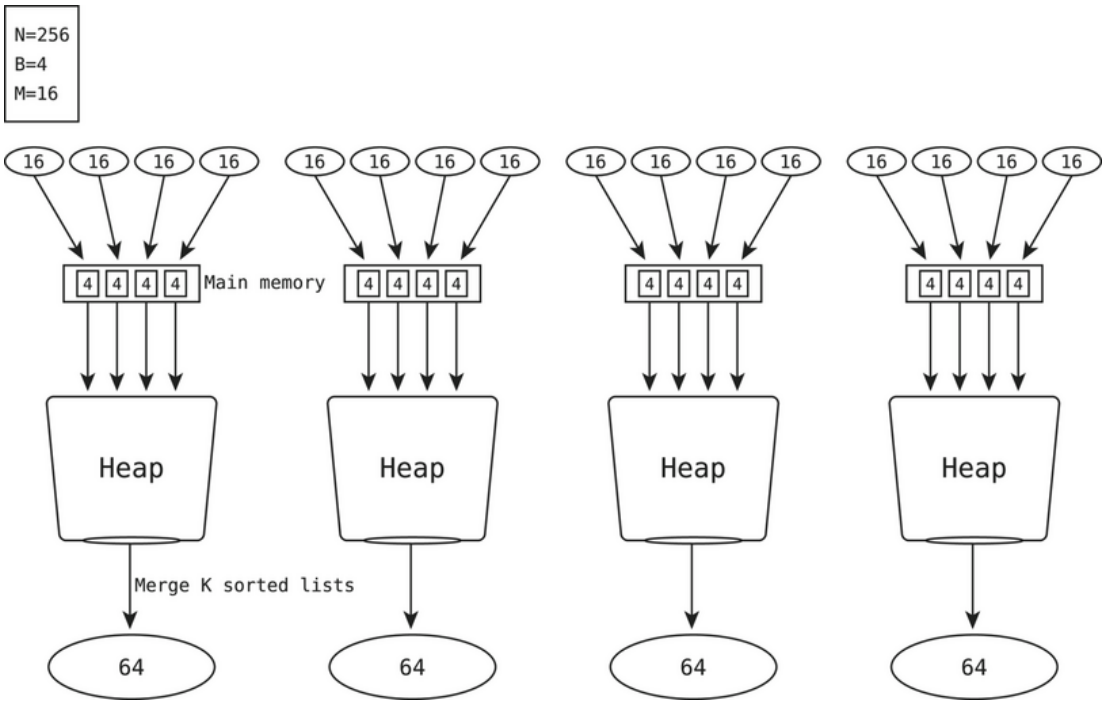
And what is that number? Well, in our scenario, it was 4, and this is because M=16 and B=4. In other words, the greatest number of blocks that RAM can hold at once is:

```
M / B = maximum number of blocks that RAM can hold
```

And because M/B is the greatest number of blocks that can fit in RAM, M/B is *also* the greatest number of sublists that we can merge at once.

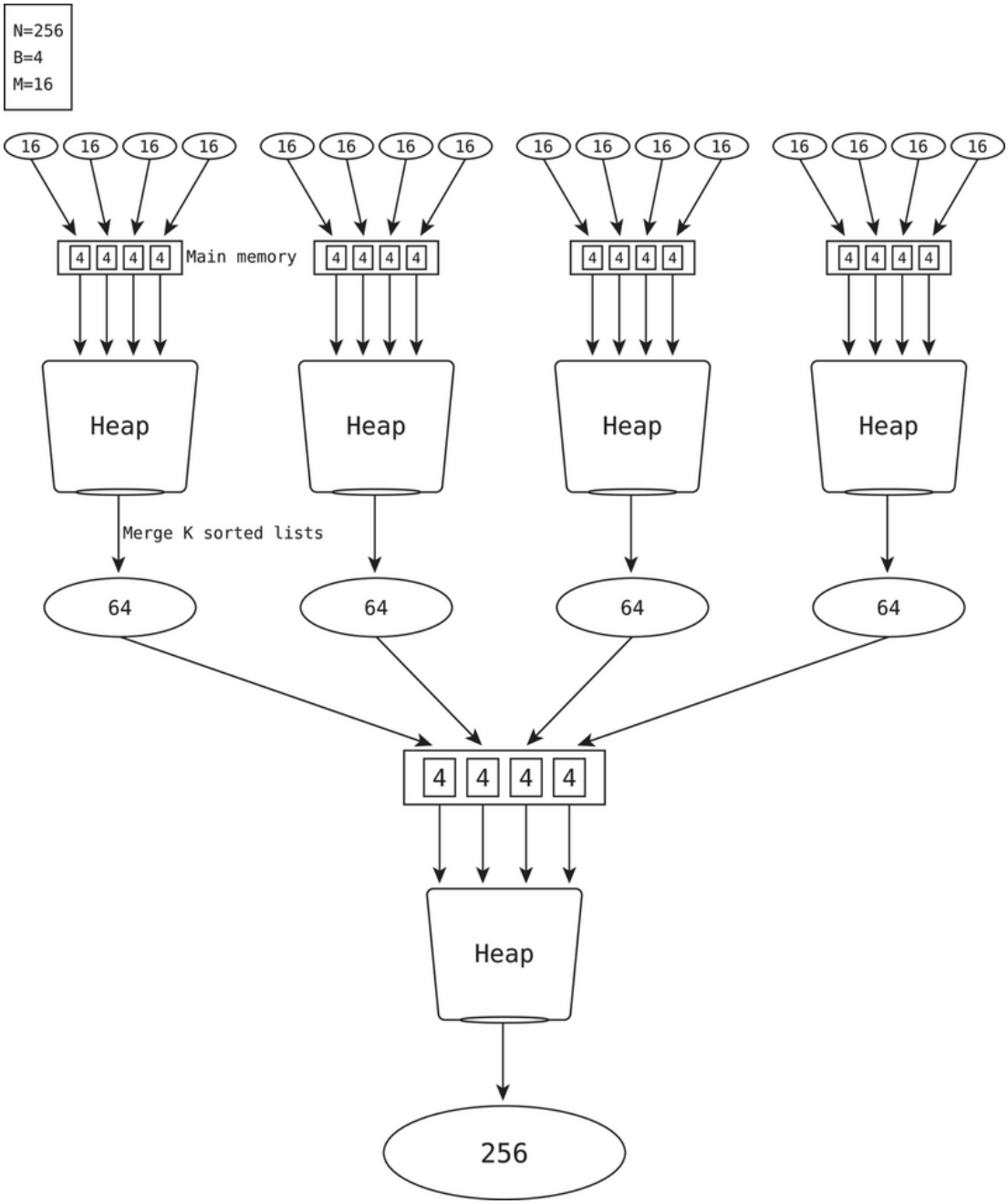
And that's where the algorithm of M/B-Way Mergesort gets its name. In M/B-Way Mergesort, we execute a number of phases in which we merge a certain number of sublists at once. And how many sublists do we merge at once? The answer is: M/B.

Let's walk through the rest of M/B-Way Mergesort. In our current example, we merged the first M/B (4) lists to create a new sublist of size 64. We then proceed to do the same with the other groups of M/B sublists:



This gives us 4 sublists of size 64 each. We're now ready for the next phase.

Once again, we will use our Top-Grade Merge algorithm to merge these 4 sublists together. And once again, we can only merge M/B sublists at once. Luckily, at this point, we only have M/B sublists to merge, so we can merge them all at once as shown in the figure.



With this algorithm, we were able to merge the entire list of 256 items in *three* phases.

As we increase the total number of values (N), the number of phases will increase. However, M/B-Way Mergesort will nonetheless remain the most efficient way to sort the values, no matter how many phases there are. Any other approach would take a greater number of I/Os.

Big O of M/B-Way Mergesort

Now that we're pros in juggling the variables N, M, B, and K, let's see if we can figure out how to articulate the Big O of M/B-Way Mergesort. Fortunately, we've already done a lot of the heavy lifting when we analyzed the Big O of Second-Attempt Mergesort. We've already figured out that the formula for calculating the number of I/Os in Second-Attempt Mergesort is:

```
number of I/Os per phase * number of phases = total number of I/Os
```

We've also already figured out that the "number of I/Os per phase" is N/B. Again, this is because in each phase we need to load all N elements, and it takes N/B blocks to do so.

This leaves us to figure out the "number of phases." As we did with our analysis of Second-Attempt Mergesort, let's skip the first phase for now.

The second phase starts with M-sized lists. As with Second-Attempt Mergesort, the number of M-sized lists we begin with is N/M. That is, if we divide N into M-sized lists, we'll have N/M such lists.

Now, in Second-Attempt Mergesort, each phase cuts the number of sublists in half. Based on this, we described the number of phases as $\log_2 N/M$. That is, each phase divides the sublists by 2 until we create one complete list.

In M/B-Way Mergesort, though, each phase divides the number of sublists by *more* than 2. In our prior scenario, each phase divided the number of sublists by *4*.

Now, we've seen that this number 4 was computed based on M/B; the maximum number of lists we can merge at once is M/B. And so, each phase consolidates M/B sublists at once. In our example, 16 sublists became 4 sublists, and 4 sublists became 1. Given that each phase divides N/M (M-sized) lists by M/B, we'd say that the total number of phases is $\log_{M/B} N/M$. In other words, we start off with N/M lists, and we keep dividing that number of lists by M/B until we get one final sorted list.

So, when we multiply this "number of phases," which we've said is $\log_{M/B} N/M$, by the "number of I/Os per phase," which is N/B, we get a final Big O of:

$O(N/B \log_{M/B} N/M)$.

If that doesn't look like gobbledygook, I don't know what does. However, this Big O Notation is useful because we can now plug in any scenario to figure out approximately how many I/Os will occur.

Let me show one quick example, specifically focusing on the improvement of M/B-Way Mergesort over Second-Attempt Mergesort. Again, the Big O of each is:

Second-Attempt Mergesort: $O(N/B \log_2 N/M)$

M/B-Way Mergesort: $O(N/B \log_{M/B} N/M)$

They're almost the same, except that Second-Attempt Mergesort has a logarithm base of 2, while M/B-Way Mergesort has a logarithm base of M/B.

Now, let's dream up a scenario where we have 100,000 data elements, a RAM size of 1,000, and a block size of 100. In other words, N=100,000, M=1,000, and B=100.

If we plug this into Second-Attempt Mergesort's Big O, we get:

```
N/B             *  log2 N/M              =
(100,000 / 100) * (log2 100,000 / 1,000) =
1,000           *  7                     =
7,000 I/Os
```

With M/B-Way Mergesort, on the other hand, we compute:

```
N/B             *  logM/B N/M =
(100,000 / 100) * (log10 100,000 / 1,000) =
1,000           *  2 =
2,000 I/Os
```

That's some pretty significant time savings right there.

Once again, pure Big O Notation is a bit limiting here. Since Big O technically drops logarithm bases, the Big O of both Second-Attempt Mergesort and M/B-Way Mergesort come out to be O(N/B logN/M). But then we wouldn't be able to tell that there's any speed difference between the two algorithms. And so, I'm going a bit on a limb and putting the logarithm bases back in so we can measure the precise advantage of M/B-Way Mergesort.

Wrapping Up

You're walking away from this chapter with a number of important tools under your belt. First, you know how to wield M/B-Way Mergesort, which is the fastest approach for using Mergesort on external memory. Second, you can now use the Top-Grade Merge algorithm to merge K sorted lists from *internal* memory in the speediest possible way. And perhaps most important, you now possess the keys for analyzing and optimizing all sorts of external-memory algorithms.

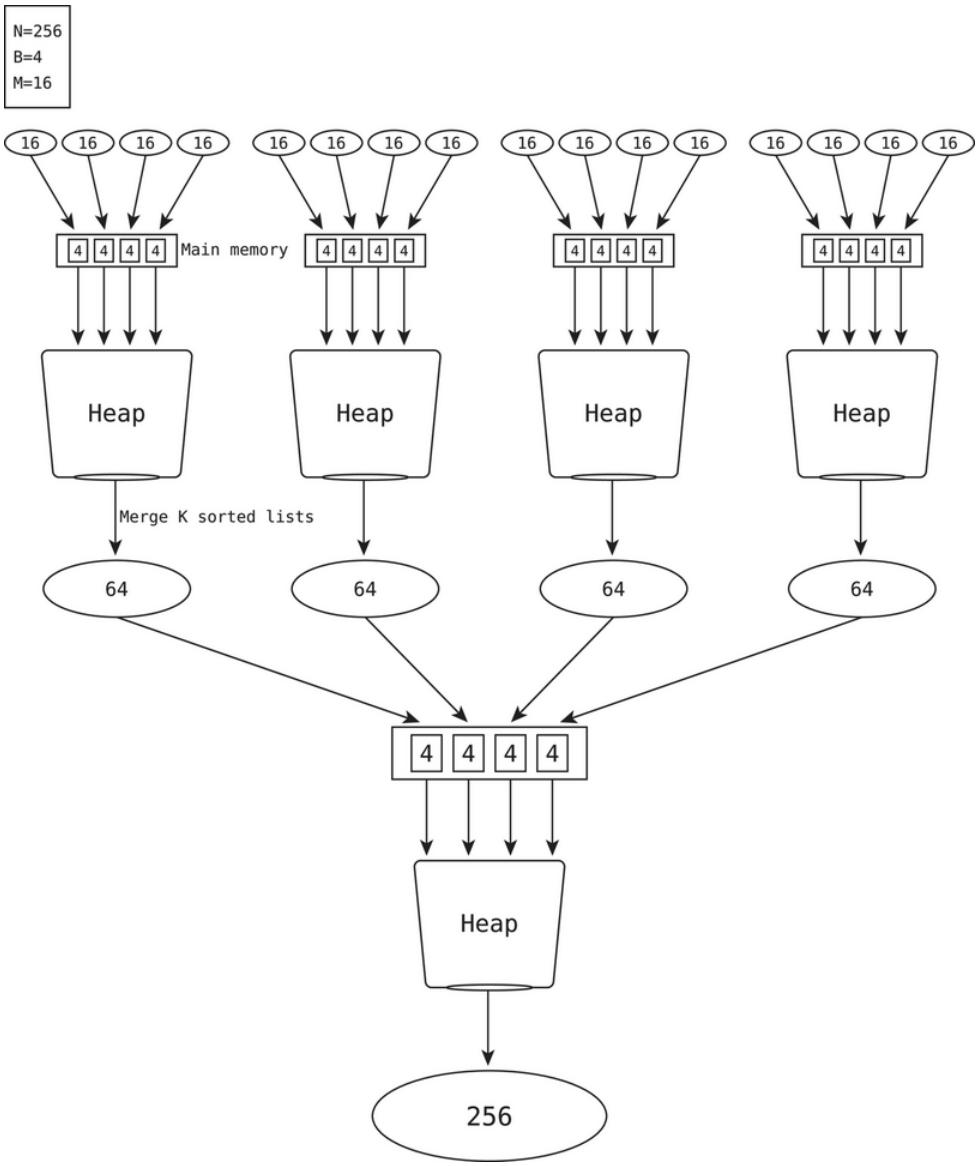
This chapter concludes our ambitious side quest into the world of external-memory algorithms. As I've mentioned earlier, this topic can fill an entire volume unto itself. So, it's time to get back to our "main" storyline: randomization algorithms.

The next chapter will reveal an entirely new class of algorithms that take advantage of randomization. These algorithms, called *Monte Carlo algorithms*, use randomization to achieve a most shocking trade-off.

Exercises

The following exercises provide you with the opportunity to practice with external-memory sorting as well as merging K sorted lists. The solutions to these exercises are found in the section *Chapter 8*.

1. Here's a diagram depicting an example of M/B-Way Mergesort:



How many I/Os take place in total?

2. How many I/Os would M/B-Way Mergesort take if we modified this scenario so that M=64 instead of M=16?

3. Say that we're conducting the Top-Grade Merge algorithm to merge data that fits entirely within memory. Specifically, we have one million values divided among 512 lists. How many steps will our merging algorithm take?

Chapter 9

Counting on Monte Carlo Algorithms

In the previous two chapters, we went on an exciting side quest and looked at the fundamentals of external-memory algorithms. But now it's time to return to the topic of randomization.

Throughout this book, I've mentioned that there's generally no one algorithm that is the "best algorithm." That is, when choosing between two competing algorithms, there's usually some sort of trade-off. Algorithm A may be faster, but Algorithm B may consume less memory. Algorithm C may be faster in the average case, but Algorithm D is faster in a worst-case scenario. Algorithm E may be better in terms of time *and* space, but Algorithm F may be simpler to implement and therefore have a smaller risk of bugs.

In this chapter, we'll take a look at a new kind of trade-off: reducing accuracy for the sake of increasing speed. Now, this may not seem to make much sense at first, but trust me; it can be the key to solving some tough problems! Let's take a look.

Monte Carlo Algorithms

I just made the audacious statement that some algorithms *reduce accuracy* for the sake of *increasing speed*. In other words, Algorithm A is slower but guaranteed to produce the correct result, while Algorithm B is faster but may not produce the correct result.

Which would you choose?

If you're a fourth-grade math teacher, this may be nothing short of terrifying. Imagine the following classroom conversation:

> You say, *"Kids, kids, settle down. Okay, who can tell me the answer to 4 times 6? Wow, Chester, your hand shot up fast! Okay, Chester, what's the answer?"*
>
> Chester blurts out, *"18!"*
>
> Some of the other children giggle softly. You say, *"No, Chester. The answer is 24. Perhaps ..."*
>
> Chester interrupts you, shouting, *"But you gotta be impressed with how fast I gave you the answer!"*

Impressive indeed. Do we want the wrong answer fast? This seems like a bad trade-off.

Yet, Monte Carlo algorithms do just this. They sacrifice accuracy for the sake of speed. (You'll see in Chapter 13, *Cultivating Efficiency with Bloom Filters* that some Monte Carlo algorithms sacrifice accuracy for the sake of saving *space*, but in this chapter we'll focus on speed.) Specifically, Monte Carlo algorithms use *randomization* to boost speed, albeit with a reduction in accuracy.

The name *Monte Carlo* comes from the Monte Carlo Casino in Monaco. In truth, though, there's not much of a connection between the casino and the similarly named class of algorithms other than the fact that both have something to do with random chance. It's a computer-science naming convention at its best.

Let's take a look at where and why we'd want to use a Monte Carlo algorithm.

Random Sampling

Let's say we're a polling agency and we want to predict the outcome of an upcoming mayoral election. If we want to get the most accurate results, we should interview *everyone* in the city who will be voting. In theory, if we managed to contact everyone who will vote, and everyone told us the truth, we'd identify our city's next mayor with certainty.

The biggest hurdle preventing us from doing this is simply the fact that *we don't have the time* (or staff) to contact everybody who lives in the city. To save time, pollsters only contact a *portion* of the total population.

This is a classic Monte Carlo algorithm at work. We save time by surveying fewer people. Although this will certainly reduce the accuracy of our poll, it may be *accurate enough* to get the job done.

Now, this only works if we conduct our poll according to the best practices of statistics. Pollsters are only okay with reducing the accuracy of the poll if statistics demonstrate that their results will likely not be far off from the truth. Accordingly, pollsters have to know how many people to interview. Obviously, interviewing one person will be super fast but will give us worthless data. Knowledge of statistics and probability is used to dictate *how* to conduct a random sampling so that our findings will likely be close to accurate.

A fundamental point to highlight about random sampling is the fact that it's, well, *random*. Let's say that statisticians determined, based on the size of the city, that we should interview 500 people. We would make a terrible mistake if we interviewed 500 people from the same neighborhood. This is because it's possible that people living in the same neighborhood have similar political beliefs or have seen the same political billboards, so we wouldn't be interviewing a group that truly represents the entire city. For sampling to work, we need to choose people at random, in which case it'll be likely that the people will be from different neighborhoods and have different demographics.

In sum, random sampling is a great example of a Monte Carlo algorithm widely used in practice. And now it makes a lot more sense as to why we'd be willing to use such an algorithm even though we're reducing accuracy. The key is that although we're reducing accuracy, we're following the rules of statistics so that our results will *likely* be accurate, or accurate *enough* for our purposes.

How Important Is Accuracy?

How accurate an algorithm needs to be depends entirely on your application. Here are some things to think about:

1. What is the worst thing that will happen if your results are *not* correct? Will you lose money? Will someone get hurt? Or is the worst thing that someone will be mildly annoyed?

2. A second item to figure out is the odds that the results will be correct, and if you're okay with those odds. For our mayoral poll, is it okay if our results only have a 95 percent chance of predicting the winner? What about 90 percent?

Here's another example to highlight these points. Let's say we're running an algorithm that will help a 10 billion dollar spacecraft land safely on

another planet. Obviously, the stakes are high. But do we need an algorithm that is guaranteed to be 100 percent accurate? What if we can produce an algorithm that's much faster but will only land the spacecraft with 99.99999999 percent probability? A careful analysis must be done to determine if we're willing to take such a chance.

Approximations

You've seen that a Monte Carlo algorithm can be, say, only 90 percent accurate. I'd like to point out, though, that this can manifest in two possible ways. One way is that an algorithm can be right 90 percent of the time, but the other 10 percent of the time, it's completely wrong. Suppose that an algorithm is supposed to produce a result of either `True` or `False`. For the 10 percent of instances where the algorithm spits out the wrong answer, it couldn't be more wrong.

However, another way in which an algorithm can be 90 percent accurate is through *approximation*. For example, suppose that an algorithm predicts tomorrow's average temperature. If the true temperature is 100 degrees, and our algorithm predicted that it would be 90 degrees, our algorithm was 90 percent accurate.

In theory, certain Monte Carlo algorithms can *guarantee* to give approximate answers that deviate no more than 10 percent of the correct answer. These "wrong" approximations yield worst-case scenarios that are potentially much less "dangerous" than results that are *completely* wrong.

Monte Carlo Algorithms vs. Las Vegas Algorithms

If you do any research on Monte Carlo algorithms, you're bound to stumble upon another term called *Las Vegas algorithms*. Like Monte Carlo, the name Las Vegas was picked simply because it's another casino location. Don't read too much into the name; casinos in Las Vegas operate just like the Monte Carlo one does.

Both Monte Carlo and Las Vegas algorithms are randomization algorithms, but have opposite natures, as follows:

- A Monte Carlo algorithm is a randomization algorithm that is guaranteed to be *fast* but has a chance, albeit a small one, of *producing the wrong answer*.

- A Las Vegas algorithm is a randomization algorithm that is guaranteed to be *correct*, but has a chance, albeit a small one, of being *slow*.

Virtually all the randomization algorithms we've dealt with up until this chapter have been Las Vegas algorithms. Preshuffling an array before inserting it into a binary search tree is a Las Vegas algorithm since the BST is guaranteed to be "accurate"—which in this context means properly structured—no matter what. We do the preshuffling to increase the likelihood that the tree will be fast by being well-balanced, but we can't guarantee that the tree will be fast. After all, we may be super unlucky and our shuffling may produce numbers that are in ascending order. In sum, this is a randomization algorithm that is guaranteed to be correct but has a small chance of being slow.

Randomized Quicksort is also a Las Vegas algorithm. No matter what, the data will be properly sorted by the time we're done. At the same time, we randomize the pivot to increase the odds that the sorting will take place quickly. However, we can't guarantee that the sorting will be fast, since we

might choose unlucky pivots. Again, this is a randomization algorithm that will definitely be correct but might run slowly in a small percentage of cases.

Monte Carlo algorithms, on the other hand, work the other way around. By sacrificing accuracy, we guarantee that the algorithm will run quickly. For example, if we decide to only poll 500 residents of the city, we guarantee that the speed of our algorithm will be whatever time it takes to poll 500 people. So, again, it's an algorithm that will definitely be fast but may not be entirely accurate.

When we think about Las Vegas and Monte Carlo algorithms further, we'll see that they utilize randomization to achieve different goals. Specifically, they use randomization to help address their own weak points.

The purpose of randomization within Las Vegas algorithms is to increase speed. Although a Las Vegas algorithm, by definition, is not guaranteed to be fast, it uses randomization to increase the likelihood of being fast. (Think about Randomized Quicksort, for example.)

Monte Carlo algorithms, by contrast, use randomization to increase accuracy. The speed guarantee of Monte Carlo is achieved by techniques like cutting corners—such as polling only 500 people instead of the entire city. However, cutting corners has the side effect of reducing accuracy. Randomization is used to mitigate this side effect and help keep the algorithm as accurate as possible.

In any case, we can now put a bow on our definition of Monte Carlo algorithms. A Monte Carlo algorithm is a randomization algorithm that is guaranteed to be fast and is likely, but not guaranteed, to be correct.

Obtaining Averages Through Random Sampling

You've seen how random sampling is used in the world of pollsters. Let's now take a look at how we can use random sampling within the world of code. We'll begin with some examples of computing averages.

Approximating the Mean

Say that we have an unsorted array containing a whole bunch of integers. To calculate the *mean* of these numbers, we first compute the sum of all N integers and then divide by N. Because we have to add up all N numbers, this algorithm takes roughly N steps.

With random sampling, though, we could calculate an *approximate* mean by picking a random sample of integers from the array and then computing the mean of that sample.

So, if our array contained 1,000,000 elements, getting the true mean would take about 1,000,000 steps. However, we could instead choose to add up 1,000 *randomly chosen integers* from the array and then divide that number by 1,000. This would be the *sample* mean. It would be less accurate than the true mean, but we'd only have to perform 1,000 steps instead of 1,000,000 steps.

Again, a proper statistical analysis needs to be done to determine if this approximate mean is good enough for your needs. But if it is, you can save a lot of time. This is especially true if your data is in the billions or trillions of elements.

For fun, though, let's play around with this idea. Here goes:

```
import random

array = []
```

```
for i in range(1000001):
    array.append(i)

random.shuffle(array)

sum = 0
random_sample_size = 500

for _ in range(random_sample_size):
    random_index = random.randint(0, 1000000)
    sum += array[random_index]

mean = sum // random_sample_size

print(mean)
```

We first build an array of the integers 0 through 1,000,000 in random order. The mean of these numbers is basically 500,000.

Next, we randomly select 500 integers from the array and compute the mean of those 500 numbers. In effect, we're computing the mean of our random sample.

When I run this code repeatedly, I get results like `501554`, `498627`, `499450`, `506332`, `477290`, `505872`, `489558`, and `496410`. These are all pretty close to the true mean of `500000`!

You can play around with the `random_sample_size` variable, which chooses the size of our random sample. Even when I reduce this to 100, the results aren't far off. I think it's pretty cool to see in action how close random sampling comes to the ultimate truth.

Approximating the Median

We can use a similar approach to approximate the *median* of an array of numbers. As a reminder, the median of a list of values is the middle-most value if the values were to be sorted. In the array, `[22, 56, 88, 89, 154, 207, 365]`, `89` is the median since it's smack in the middle. This would be the case even

if the array were shuffled; the point is that the median is the center value once the values are ordered.

If there are an even number of values, there's no single value in the center. Rather, there are *two* values in the center. When this is the case, the median is the mean of those two values.

In the following code, I use random sampling to approximate the median of the numbers 0 through 1,000,000. The median in this case happens to be the same as the mean—it's 500,000. Note that the approach is similar to what we did for approximating the mean:

```
import random

array = []

for i in range(1000001):
    array.append(i)

random.shuffle(array)

sum = 0
random_sample_size = 501
random_sample = []

for _ in range(random_sample_size):
    random_index = random.randint(0, 1000000)
    random_sample.append(array[random_index])

random_sample.sort()
median = random_sample[random_sample_size // 2]

print(median)
```

Here, we grab a random sample of 501 elements (I made the sample size odd to keep things simple). We then compute the median of the random sample by first sorting the random sample and then grabbing the value at the center index, which is `random_sample_size // 2`.

When I run this code, I also get results that are close to the truth, although not as close as when I approximated the mean. Try it out for yourself!

Primality Testing

One classic application of Monte Carlo algorithms is primality testing. *Primality testing* means to test a number to determine whether it's a prime number. Finding prime numbers is not just a trivial math concept; it's important in many contexts, particularly in the field of cryptography.

In case you need a quick refresher, a prime number is one that cannot be divided by any other whole number (save for the same number itself or the number 1) without leaving a remainder. The number 17, for example, is prime because every number we try dividing it by will leave a remainder. The fact that 17 *can* be divided by 1 and 17 doesn't rule 17 out from being prime, since *every* number can be divided by itself or the number 1.

Another way of defining a prime number is that a prime number is one in which no two whole numbers (that are both greater than 1) multiplied together produce that number.

The opposite of a prime number is a *composite* number. A composite number *can* be divided by another number (that isn't 1) without leaving a remainder. The number 15 is a composite number because it can be divided by 3 or 5 without leaving a remainder. Said another way: because we can find two whole numbers that multiply together to produce 15, namely the numbers 3 and 5, the number 15 is composite.

Perhaps the most straightforward way to test whether a number is prime is by using brute force. That is, we'll try dividing the number by every other smaller whole number. So, if our number is 11, we'll first try dividing by 2, and then by 3, and then by 4, and so on. As soon as we find a division that doesn't produce a remainder, we would conclude that 11 is composite. However, if we try all the possible divisions and can't get a remainder (which indeed is the case with the number 11), we'll know that the number 11 is prime.

In Volume 1, Chapter 3, we wrote a function that does this. Here it is:

```
def is_prime(number):
    for i in range(2, number):
        if number % i == 0:
            return False

    return True
```

Here, our function accepts the `number` we're testing for primality. We run a loop that divides the `number` by every integer `i` from 2 up until the `number` itself. If we find one such division that leaves no remainder, we return `False` to indicate that the number is *not* prime. However, if we get through the entire loop without finding a "remainderless" division, we return `True` since the `number` must be prime.

Improving Brute-Force Primality Testing

We can optimize our `is_prime` function in a few ways. As things stand now, we take N steps to run this primality test, if we consider `number` to be N. That could take a long time if we're testing whether the number 20,988,936,657,440,586,486,151,264,256,610,222,593,863,921 is prime! (It is.)

One simple trick we can do to speed things up a bit is to first check if `number` is even. If it is, `number` is certainly composite since it can be divided by 2. (The exception is if `number` is itself 2; then it's prime.)

With even numbers out of the way, our loop only has to divide `number` by odd integers. Here's the code:

```
def is_prime(number):
    if number == 2:
        return True

    if number % 2 == 0:
        return False
```

```
    i = 3
    while i < number:
        if number % i == 0:
            return False

        i += 2

    return True
```

With this easy trick of skipping even numbers, we've effectively shaved off half our search time.

Now, if we think about things a bit more (it's always good to think about things), we'll find that we can speed up our primality test even further. To help paint the picture, I'm going to refer to some division jargon—the type of stuff we learned about in fourth grade but subsequently forgot:

In the division operation `36 / 9 = 4`, `36` is the *dividend*, `9` is the *divisor*, and `4` is the *quotient*.

An important property of division is that if `36 / 9 = 4`, then it's also true that `36 / 4 = 9`. This is because `4 * 9` and `9 * 4` both equal `36`. In other words, a division equation holds true even after we swap the divisor with the quotient.

Now, let's say that I want to test whether the number `37` is prime. Let me begin:

```
Dividend        Divisor        Quotient        Remainder
37       /        2     =          18               1
37       /        3     =          12               1
37       /        4     =          9                1
37       /        5     =          7                2
37       /        6     =          6                1
```

Here, I tried dividing `37` by the numbers `2` through `6`. The quotients range from `18` down to `6`. Now, watch what happens when I divide `37` by `7` and `8`:

```
Dividend        Divisor        Quotient        Remainder
37       /        7      =          5               2
37       /        8      =          4               5
```

Note that at this point, the quotients we're getting now *are the same as the divisors from our first bunch of division computations*. What this means is that I don't have to bother dividing `37` by `7` or `8` to see whether `37` is prime. After trying to divide `37` by `6`, from here on in, I'm only going to get quotients that lie in the range of divisors that *I already tried previously*. Accordingly, I know that if I couldn't find any remainderless quotients earlier, there's no way I'm going to find any remainderless quotients now. And so I can conclude that `37` is prime.

This inflection point occurs when the divisor reaches the *square root* of the dividend. As a reminder, because `6` squared is `36`, we say that `6` is the square root of `36`.

When I divide `36` by `2` and `3` and so on, the quotients are still new numbers we haven't encountered before as divisors. This changes after I divide `36` by its square root, which is `6`. Henceforth, all the quotients we'll get by dividing by `7` and on will all lie in the range of divisors we've already tried before. The same applies to the `37` example since the `6` is the approximate square root of `37`.

With this all in mind, we can now optimize the `is_prime` function to only run its loop up until the square root of `number`:

```
import math


def is_prime(number):
    if number == 2:
        return True

    if number % 2 == 0:
        return False
```

```
    i = 3
    while i <= math.sqrt(number):
        if number % i == 0:
            return False

        i += 2

    return True
```

This algorithm belongs to a category of Big O notation we haven't encountered before in this book: O(sqrt(N)). An algorithm is described as O(sqrt(N)) when the algorithm takes sqrt(N) steps when there are N elements of data. In our case, the algorithm takes only half that number of steps since we're only dealing with odd numbers. Although this is truly sqrt(N)/2 steps, we drop the constant, leaving us with O(sqrt(N)).

These are some pretty clever optimizations, so we should definitely pat ourselves on the back. However, even this optimized algorithm is still no match for massive numbers. The square root of a massive number may be a massive number itself. As such, our primality test will still be impossibly slow.

And that's where Monte Carlo algorithms come in.

Monte Carlo Primality Testing

I'm going to take a stab at proposing a Monte Carlo algorithm for primality testing that will take a trivial amount of time. It'll turn out to be a terrible algorithm in practice, so don't try it at home! In fact, I'm going to call this method "Bad Primality Testing." I'm only introducing it because it'll make the "good" primality testing easier to understand.

Here is the Bad Primality Testing algorithm:

Instead of dividing number by all odd integers from 3 and up until the square root of number, we'll divide number by a bunch of *randomly chosen* integers. If we choose, say, 100 random integers and divide number by them and always get a remainder, there's a certain probability that number is prime.

This certainly qualifies as a Monte Carlo algorithm. It has a guaranteed speed, as we perform a fixed number of 100 division operations. However, it only has a certain likelihood of being correct.

The problem with Bad Primality Testing, though, is that it's not reliable enough. For example, many composite numbers have *only one pair* of smaller numbers that multiply into that composite number. Indeed, this is what happens when the smaller numbers are themselves both prime. So, if we test a number like this, it's likely that none of our 100 random divisions will divide our number by either of those smaller numbers that compose our number. And so, the odds are high that we'll mistakenly identify our number as prime when it's composite.

If our Monte Carlo algorithm will likely be wrong for certain numbers, it's a pretty poor algorithm. Monte Carlo algorithms are valuable when the odds are that they'll be correct, even if there's no guarantee. But if the odds are in favor of getting the *wrong* answer, we'd better avoid such an approach.

Fermat's Little Theorem

Thankfully, there are *good* Monte Carlo algorithms for primality testing. A number of them revolve around a theorem first published back in the year 1640.

Pierre de Fermat, a 17th-century French mathematician, came up with a number of mathematical theorems in his lifetime, and the one relevant to us is known as *Fermat's Little Theorem*. (Yes, it's actually called that, and don't ask me why.)

Fermat's Little Theorem can be expressed in a number of ways, but I'll present it in the way that I feel is clearest for our context. Here goes.

Let's use the variable N to refer to a number we're testing for being prime. Every number N has a series of smaller numbers that run from 1 up to N, excluding N itself. We're going to use another variable, A, to refer to these smaller numbers.

Now, it may sound strange that a single variable can refer to a bunch of numbers, but here's what I mean. If N is 7, the numbers 1, 2, 3, 4, 5, and 6 all qualify as A. So, if we have a formula that contains our A variable, we can plug any one of the numbers from 1 through 6 into A that we'd like; it's our choice. This will become clearer in a moment.

Fermat's Little Theorem is that if N is prime, then the following formula (expressed in Python code) will be true for all possible numbers that we can plug into A:

```
A**(N-1) % N = 1
```

This expression states that if we take a number `A`, raise it to the power of `N-1`, and then divide the result by `N`, we'll get a remainder of `1`.

In other words, what Fermat's Little Theorem is saying is: if N is prime, then *all* possibilities of A will plug into this formula and produce a result of 1.

This may sound confusing, I know. So, let me clarify this theorem with an example. For the rest of this chapter, I'm going to refer to `A**(N-1) % N` as "The Formula." (It's way easier to type.)

Let's say that N is 7, which is prime. The list, A, as mentioned, includes the numbers 1, 2, 3, 4, 5, and 6. If we take any number from A, say the number 3, and apply The Formula, we get:

```
A**(N-1) % N =
3**6 % 7 =
1
```

As you can see, this computes to 1.

Fermat's Little Theorem says that The Formula will compute to 1 for *all integers that qualify as A*. To demonstrate, all of the following statements are true:

```
1**6 % 7 = 1
2**6 % 7 = 1
3**6 % 7 = 1
4**6 % 7 = 1
5**6 % 7 = 1
6**6 % 7 = 1
```

You have to admit that this is pretty cool. We're not going to look at why this is so, but let's run with it.

To recap, Fermat's Little Theorem claims that if N is prime, then for all of A (which are the numbers from 1 up until N, not including N), if we compute The Formula, we'll get the result of 1.

Now, the following statement is key, so listen up and listen well.

A logical equivalence of Fermat's Little Theorem is that if we compute The Formula and get a result that is *not* 1, then we know that *N is composite*. This follows logically, for if we're guaranteed that for a prime N that The Formula will produce 1, that means if we *don't* get a result of 1, then N cannot possibly be prime.

What Fermat's Little Theorem does *not* mean, though, is that if we do get 1, then N is prime. Many composite numbers have at least one number A in which The Formula will produce 1. All Fermat said was that if N is prime, then The Formula will produce 1 for *every A*. And equivalently, if The Formula does not produce 1 for any given A, then we know that N is composite.

It turns out that Fermat's Little Theorem is a *one-directional* rule. We can use it to prove whether a given number is composite, but cannot use it to prove whether the number is prime.

In any case, now that we're armed with Fermat's Little Theorem, we're ready to discover a *good* Monte Carlo primality test.

Fermat's Primality Test

Remember our Bad Primality Testing algorithm? We chose 100 random numbers and divided a `number` by them to see if `number` was prime. This turned out to be a losing proposition. However, we *can* use a similar tactic that utilizes Fermat's Little Theorem. This approach is called *Fermat's primality testing* since it's predicated upon Fermat's Little Theorem. I'm going to unfold this algorithm one logical layer at a time, so bear with me.

We've already derived logically from Fermat's Little Theorem that if we take a number N and run it through The Formula by plugging in some example of A and do not get a result of 1, then N must be composite.

Based on this, we can pick 100 random numbers to plug into the A variable, and run each example of A through The Formula. So, if N is 45,321, we'll randomly pick 100 different example numbers for A, such as 3,453, 19,001, and 767, and test them out. If any of them produce a result that isn't 1, we'll know that N is composite.

However, if we get a result of 1 for all 100 tests, *there's a high likelihood that N is prime*. This is because a composite number is unlikely to pass all 100 tests. (Soon, we'll discuss precisely what the odds are.)

That being said, even when it passes 100 tests, N may in reality be composite. That is, N may in truth have some instances of A where the result of The Formula is 1 and other results where the result is *not* 1. However, we happened to be super unlucky and picked all the examples of A where The Formula will yield a 1. So, we'd mistakenly identify N as being prime even though it's composite.

As such, this approach to primality testing is a Monte Carlo algorithm. The odds are high that N is prime, but there's a small chance that it's composite. Note, however, that the chance of error can only happen in one direction.

That is, if The Formula ever produces a result other than 1, we can be 100 percent sure that N is composite. If, on the other hand, The Formula produces 100 instances of 1, although there are high odds that N is prime, we can't be absolutely certain that it is so.

Fermat's vs. Bad Primality Testing

Let's dig in deeper. Why, exactly, is Fermat's Primality Testing any better than our previous approach of Bad Primality Testing? After all, in both techniques we try out 100 computations using random numbers.

Here's the answer. Let's look again at what happens when we perform Bad Primality Testing. Say that we're testing whether the number 47,957 is prime by dividing it by a smaller random number, such as 45, to see if there's a remainder. If this doesn't produce a remainder, we'd know that 47,957 is composite. But on the other side of the coin, if it *does* produce a remainder, what does that tell us about the probability of 47,957 being prime?

In truth, 47,957 is only divisible by the numbers 217 and 221. Indeed, *many* composite numbers are only divisible by a couple of smaller numbers. So the fact that we got a remainder when dividing 47,957 by 45 tells us almost *nothing* about the odds of whether 47,957 is prime. We've only eliminated one possible way in which 47,957 could be composite, but there may easily be other numbers that 47,957 is divisible by. And so, we've hardly moved the needle in seeing increased odds that 47,957 is prime.

However, each computation in Fermat's Primality Testing that produces a 1 does significantly increase the odds that the number we're testing is prime. Fermat never articulated the following statement, but other mathematicians did. This statement is the final piece of the puzzle:

> *If N is a composite number, each time we run The Formula, the chance of getting a result of 1 is 50 percent or less.*

Let that sink in for a moment.

Recall that I pointed out that Fermat's Little Theorem is one-directional. That is, if we run The Formula on a prime number N, there's a 100 percent chance that we will get a result of 1. If N is composite, though, it may or may not produce a 1.

However, we're filling in an additional detail now by saying that if N is composite, the odds of us getting a 1 for any time we run The Formula are 50/50. In truth, the mathematicians stated that the odds may be less than 50/50, but let's call it 50/50 to simplify things.

Let's see how this all plays out. If I test N against one example of A, and get a 1, I haven't learned much. After all, there's a 50/50 chance that N is composite.

But say I test N again.

And again.

And again.

This is like flipping a coin. It's not remarkable when a coin lands on heads; there was a 50 percent chance that this would happen. But if I flip a coin *many* times and always get heads, that's truly remarkable since such a result is very unlikely.

With Fermat's Primality Testing, when we test N against 100 examples of A, it's like flipping a coin 100 times. Sure, if N is composite, each "flip" has a 50 percent chance of yielding a 1. But if we make 100 flips for a composite number, it's *extremely unlikely* that all 100 flips will produce a 1. This is like flipping a coin 100 times and always getting heads. (In math terms, we'd say the odds are $1/2^{100}$.)

This is why Fermat's Primality Testing is a highly effective algorithm. If we run The Formula 100 times on N and always get a 1, it is highly probable that N is prime.

And this, my friends, is how Fermat's primality testing works. To sum it up:

To test N, we pick 100 random examples of A and run The Formula for each example. If we ever get a result that is not 1, we'll know with 100 percent certainty that N is composite. And if we *always* get a 1, then it's highly probable that N is prime.

Fermat's primality testing is a solid Monte Carlo algorithm because it has a guaranteed speed of 100 tests, but it is not necessarily correct. However, because it's highly probable to be correct, this algorithm can produce results that you might consider to be accurate *enough* for your particular application.

There's one tiny caveat, though. We've gone through all the logical hoops of analyzing Fermat's Little Theorem, except for one, which we'll take a look at next.

Carmichael Numbers

Fermat asserted that if N is prime, if we run each and every example of A through The Formula, we'll always get 1. We've deduced from this that if we ever get a result that isn't 1, then N must be composite. We also saw that if we do get a result of 1, we can't know for certain whether N is prime or composite. Even a composite number has a 50/50 chance of producing a 1 for each example of A.

But what happens if we methodically run The Formula for each and every A and always get 1? Fermat said that if N is prime, we'll always get 1, but let's analyze the converse. If we always get 1, does that mean that N is prime?

It turns out that it does *not* mean that N is certainly prime. For there are some composite numbers that will produce 1 each time we run The Formula —even for all instances of A.

These special composite numbers are called *Carmichael Numbers*, named after the mathematician Robert Carmichael, who researched these numbers in depth.

The smallest Carmichael Number is 561. The number 561 is composite, as 561/17=33. However, incredibly enough, if we run The Formula for all examples of A (that is, 1 through 560), the result will always be 1.

Carmichael Numbers throw a nice little monkey wrench into Fermat's Primality Testing. The success of this testing relies on our assertion that the odds of The Formula computing a 1 for a composite number are no greater than 50 percent. However, this doesn't hold true for Carmichael Numbers, as the chance of computing a result of 1 is 100 percent!

This being said, Fermat's primality testing is still useful because Carmichael Numbers themselves are rare. From numbers 1 up until 25,000,000,000, there are only 2,163 Carmichael Numbers. Additionally, the frequency of Carmichael Numbers drops as we deal with higher and higher numbers.

In other words, we kind of incorporate the rarity of Carmichael Numbers into our Monte Carlo algorithm. That is, for a "regular," non-Carmichael composite number, there's a high probability that our 100 tests will reveal that it's composite (by producing a number other than 1). And even though this isn't the case for Carmichael Numbers, there's a high probability that our number simply isn't a Carmichael Number.

If you've decided that Carmichael Numbers have the potential to mess up your application, you're still in luck. There are other primality testing algorithms—which are *extensions* of Fermat's Primality Testing—that

properly handle the edge case of Carmichael Numbers. Some such algorithms include the Miller–Rabin test and the Solovay–Strassen test. If this piques your interest, go check them out.

Code Implementation: Fermat's Primality Test

The code for Fermat's primality test is concise and simple:

```
import random


# Fermat's Primality Test
def is_prime(number):
    for _ in range(100):
        a = random.randint(1, number - 1)
        if pow(a, number - 1, number) != 1:
            return False

    return True
```

To test the primality of `number`, we run a loop 100 times. In each iteration, we choose a random number `a`, representing what we've been calling "A"—which is the series of smaller numbers from 1 up until `number`.

Next, we compute The Formula to see if we get a remainder of `1`. To accomplish this, we use the code:

```
if pow(a, number - 1, number) != 1:
```

This syntax may not be familiar to you, so here's a brief explanation:

It turns out that Python has at least five different ways that you can calculate exponents. The most popular approach is the `**` operator. If we take this approach, we can run The Formula using the code:

```
if a**(number - 1) % number != 1:
```

However, it turns out that the `**` is unable to compute exponents using very large numbers. While this code worked fine when I tried to test the integer `563` for being prime, it flatly refused to cooperate when the `number` was `2147480219`.

However, Python has other ways to compute exponents, including the built-in `pow` method. And fortunately, `pow` is able to process large primes like `2147480219`.

The `pow` method accepts a minimum of two arguments, the first being the base number and the second being the exponent. However, `pow` comes with a super-convenient feature. It accepts an optional *third* argument that takes the result of computing the first two arguments and divides it by the third argument modulus style, giving us the remainder. In other words, `pow(x, y, z)` is the equivalent of `x**y % z`. And that's exactly what we want!

If the result of this calculation is not `1`, it means `number` is composite, so we return `False`. However, if after running The Formula 100 times we always get a result of `1`, we assume with high probability that `number` is prime and so we return `True`.

Wrapping Up

Monte Carlo algorithms have the ability to greatly boost the speed of operations that might otherwise take a *really* long time. Of course, they come at the cost of some accuracy. But if you'll get the correct result with high probability, it can sometimes be a worthwhile trade-off. Ultimately, how much you need perfect accuracy will depend on your specific application.

In the next chapter, we're going to look at another randomization topic known as *randomized hashing*. With it, we'll take our understanding of hash tables to the next level. (Eventually, in Chapter 13, we'll combine the concepts of Monte Carlo algorithms and randomized hashing to produce an astounding data structure known as a *Bloom Filter*.)

Ready to dive deeper into how hash tables work and how randomization helps them become more efficient? Well, turn the page.

Exercises

The following exercises provide you with the opportunity to practice with Monte Carlo algorithms. The solutions to these exercises are found in the section *Chapter 9*.

1. Say that we have a large array of integers, and we want to determine what percentage of the integers are even and what percentage are odd. Write code that performs random sampling to approximate these percentages.

2. Let's say that I ran The Formula from Fermat's Primality Test on an integer N a total of four times. Three times the result was `1`, and one time the result was `2`. What are the odds that N is prime?

3. Now, let's say that I ran The Formula from Fermat's Primality Test on an integer N a total of four times, and all four times the result was `1`. What are the odds that N is prime?

Chapter 10

Designing Great Hash Tables with Randomization

Hash tables are ubiquitous, and for good reason. I first introduced hash tables and demonstrated how they worked in Volume 1, Chapter 8. As I discussed there, hash tables are blazing fast, as they have O(1) search, insertion, and deletion. While this speed comes at the cost of not being able to keep its values in a sorted order, hash tables are nonetheless one of the most important data structures out there.

In this chapter, we're going to revisit the inner workings of hash tables, especially regarding how *hash functions* work. In particular, we're going to take a look at a sneaky, easily unnoticed vulnerability of hash tables and how we can use randomization to make things better.

Hash Functions: A Quick Review

Under the hood, a hash table stores its data inside an array or similar structure. However, while an array usually inserts new data at its end, a hash table uses another approach in how it decides where to insert each piece of data. We covered this concept back in Volume 1, but here I'll remind you of the most pertinent details and then take the discussion further.

To keep things simple, I'll use examples of inserting integers into the hash table. (You can explore inserting strings, though, in the *exercises* of this chapter.) Also, although hash tables generally store key-value pairs, I'm going to keep my examples simple by making the key and value the same piece of data. That is, if I say that I'm inserting the integer 17 into the hash table, both the key and value will be 17.

Now, let's work with an example hash table that uses an array of size 10 under the hood. This means we're storing each piece of data into one of 10 slots:

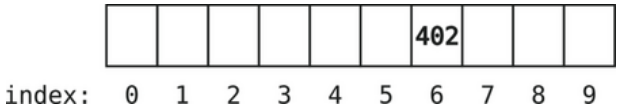
To decide which slot each value will go in, a hash table uses something called a *hash function*. A hash function is a function that converts a value into some number, known as the *hash code*. When we insert a value into the hash table, the hash table computes that value's hash code, and then inserts the value into the *index that matches that hash code*.

Let's look at an example of a simple hash function that works as follows: we take the value we're inserting and, assuming it's an integer, add up the sum of all its digits to produce a hash code.

For example, say we want to insert the integer 402 into the hash table. The hash function takes the 402 and adds up its digits:

4+0+2=6

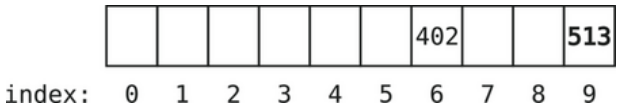
This produces 6 as the hash code. Because of this, we store the 402 at index 6:



Next, say we want to insert 513. The hash function transforms the 513 into the hash code:

5+1+3=9

Because we get a hash code of 9, we insert the 513 into index 9:



Running a value through a hash function is known as *hashing* the value. In short, hashing transforms a value into an integer—namely, the hash code.

The power of hash tables lies in the fact that *the value itself* determines where the value is going to be stored. And this is precisely why hash-table search takes just O(1) time. To search for the value 513 in the future, we run it through the hash function, get the hash code of 9, and immediately know that we can find 513 at index 9.

Now, there's an itty-bitty problem with our proposed hash function. That is, say that we want to insert the value 183.

When we hash it, we get:

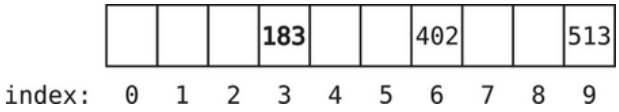
1+8+3=12

This means we'd store 183 in index 12. But there *isn't* an index 12 in our example hash table; the highest available index is 9.

To fix this, we can add another detail to our hash function: if the hash code has more than one digit, we then add *those* digits until we produce a hash code that is only a single digit. So, in this case, we'd hash the hash code of 12:

1+2=3

So, we'd insert 183 at index 3 of the array:



Scalable Hash Functions

You've now seen how important it is that a hash function aligns with the size of a hash table's underlying array. Now, determining the size of the array depends primarily on the number of values you expect to insert. In Volume 1, Chapter 8, I discussed the recommendation that a hash table should have 10 slots for every 7 pieces of data. (There are other recommendations out there, as well as other factors to consider when making this decision, but I won't get into the nitty-gritty details here.)

Once you determine the array size, you need to carefully choose a hash function that will distribute values uniformly across the array.

If, for instance, we decided that our underlying array should have 500 slots, the hash function we used earlier would be downright terrible. After all, that function can only possibly insert values into indexes 0 through 9! Our remaining 490 array slots would never be used.

In truth, even if a hash function distributed values across the length of the array but couldn't—for whatever reason—hash a value into the number 79, this hash function is considered to be deficient. In computer science jargon, we'd say that such a hash function lacks *uniformity*. The ideal hash function should be designed so that each slot in the array will likely contain the same number of items, or at least close to it.

So, how do you come up with a hash function that caters to the size of the array and ensures uniformity?

The Division Method

There are many hash functions out there that are considered effective. In this chapter, we'll focus on one particular hash function design. It's simpler than most of the alternative approaches, but still effective.

The hash function I'll demonstrate doesn't have a fancy name, but is often called *Division Hashing* or the *Division Method*. There are several variants and optimizations of this method, but I'll present it in its most basic form.

Before we start, let's define a couple of variables. We'll use the variable K to refer to the integer we're hashing, and M will refer to the size of the array. Although I personally would have chosen other alphabet letters, these are the variables that are often used in the literature.

Division hashing takes K and applies the following formula:

```
K % M
```

In other words, we divide the integer we're hashing by the size of our array. Yes, that's it.

Let's see how this plays out with some examples. Say that M is 89. If K (the integer we're hashing) is 1632, we get:

```
1632 % 89 = 30
```

That is, when we divide 1632 by 89, we get a remainder of 30. So, 30 is our hash code.

This is an effective hash function, as it caters to the size of the array *and* is uniform. Let me explain.

When we divide a number by 89, the resulting remainder is guaranteed to be some integer from 0 up through 88, inclusive. Now, this is *perfect* for our

underlying array, which has that same range of indexes, namely, 0 through 88. So, it makes a lot of sense to hash K by simply dividing it by the array size (M) since the remainder will match one of our array's indexes.

Also, the distribution is uniform, and here's why. In our example, K was 1632, and we got a hash code of 30. Let's see what happens when we keep increasing K by 1:

```
1633 % 89 = 31
1634 % 89 = 32
1635 % 89 = 33
1636 % 89 = 34
1637 % 89 = 35
```

Let's skip a few steps to where K is 1688:

```
1688 % 89 = 86
1689 % 89 = 87
1690 % 89 = 88
1691 % 89 = 0
1692 % 89 = 1
```

Once K is 1691, we get a remainder of 0, and then start the cycle again. So, if we were to insert all the integers from 0 to 1690, they'd be uniformly distributed across indexes 0 through 88. The same applies to all values greater than 1690 as well.

M Should Be Prime

One commonly recommended optimization to the Division Method is that we should make sure that M is a prime number. Even if we've determined that the size of our array needs only to be 10, it is usually worth it to increase M to 11 since 11 is prime.

Without getting too much into number theory, the basic reason for this is a concern that if M is *not* prime, our data may follow a nonuniform pattern. This happens in particular if all instances of K (that is, all the items in our

data set) and M are both divisible by some other number, such as 12 and 9 being both divisible by 3.

Here's an example that highlights this concern. Say that our data consists exclusively of even integers. This isn't so hard to imagine; perhaps we have a list of test scores where it was only possible to get an even-numbered score.

If each test score is even, this means that each score is divisible by 2. And if M is, say, 10, then M is *also* divisible by 2.

Now, let's also say that our test scores are **[2, 4, 6, 8, 10, 12, 14, 16]**. Here's what happens when we hash them when M is 10:

```
2 % 10 = 2
4 % 10 = 4
6 % 10 = 6
8 % 10 = 8
10 % 10 = 0
12 % 10 = 2
14 % 10 = 4
16 % 10 = 6
```

It turns out that we'll only place values in even-numbered indexes of the array. The odd slots of the array will never be used. As we've learned, this hinders the effectiveness of a hash function.

Let's look at one more example. Say that our test scores are all divisible by 3, such as **[12, 15, 18, 21, 24, 27, 30]**. If M is 9, which is also divisible by 3, our hash codes end up being:

```
12 % 9 = 3
15 % 9 = 6
18 % 9 = 0
21 % 9 = 3
24 % 9 = 6
27 % 9 = 0
30 % 9 = 3
```

Yikes! We'll only be storing values in the array's 0, 3, and 6 indexes. Most of the array will never be used.

We can solve much of this issue by making M a prime number. If M is prime, we can *never* have the issue where the data and M are both divisible by the same third number, for a prime number isn't divisible by *any* other number!

And so, if we make M prime, such as 11, we'll see that our integer distribution becomes uniform:

```
12 % 11 = 1
15 % 11 = 4
18 % 11 = 7
21 % 11 = 10
24 % 11 = 2
27 % 11 = 5
30 % 11 = 8
33 % 11 = 0
36 % 11 = 3
39 % 11 = 6
42 % 11 = 9
```

Every single hash code from 0 through 10 is computed once.

The same holds true for other data patterns, such as with the earlier example of even-numbered test scores:

```
2 % 11 = 2
4 % 11 = 4
6 % 11 = 6
8 % 11 = 8
10 % 11 = 10
12 % 11 = 1
14 % 11 = 3
16 % 11 = 5
18 % 11 = 7
20 % 11 = 9
22 % 11 = 0
```

It emerges that making M prime helps distribute data uniformly even when the data follows certain patterns. Now, don't forget that M represents the size of the array. So, this means that we're making sure that the array size itself is prime.

However, this trick of making M prime doesn't solve the uniformity issue for *all* patterns.

Randomized Hashing

Despite our best efforts, it's *still* possible to have a data set in which a given hash function will not distribute the data with ideal uniformity. Let's look at some examples.

Example 1: If M is 11, and our data is **[55, 22, 99, 88, 11, 66, 44, 77, 33]**, all the data will end up at the array's index 0 since all these integers of our data set are divisible by 11. We've mentioned that uniformity gets messed up when the data set and M are both divisible by a common third number. However, it's also problematic if all the data is divisible *by M itself*.

Example 2: Let's keep M at 11, but use the data set of **[56, 23, 100, 89, 12, 67, 45, 78, 34]**. This data set is the data set from Example 1, except that each integer has been increased by 1. When we divide each of these numbers by 11, we get a common remainder of 1.

Example 3: Even if the data were a mix of Examples 1 and 2, all of the data will hash into either 0 or 1. This is pretty bad, as we'll be using just 2 out of the 11 available slots in the array. That is, all data would end up at index 0 or index 1 and not anywhere else.

One might brush this off by claiming that the odds of having these types of data sets are slim. Indeed, that may be true if the data were picked randomly. However, as I pointed out earlier, it's not unreasonable to have exam scores that are all divisible by the same number.

Furthermore, there are security issues to consider. If we have a high-volume web application that depends on a hash table to allow for lookup speeds of O(1), a nefarious hacker may be able to pull the rug out from under us. If the hacker knows precisely what hash function we're using, the hacker can purposely feed our app data where all the data ends up in the same slot of the underlying array. This could slow down our app to the point where we

can only search in O(N) time, and our app might conk out due to the heavy load.

In short, computer scientists are horrified by the possibility of a data set on which a given hash function performs poorly.

Fortunately, there's a solution to this problem. And once again, randomization comes to the rescue.

The concept of *randomized hashing* says that when we create an instance of a hash table, we *randomly pick the hash function* that the hash table will use for the remainder of the hash table's existence.

I mentioned earlier that there are many different hash functions out there. Now, each hash function has a weak spot, which is the particular data set for which the hash function will not distribute the data uniformly. The idea behind randomized hashing, though, is that each hash function has a *different* weak spot. So, even if we have a data set that won't work well with Hash Function #1, it'll work out fine if we instead use Hash Function #2.

If we have, say, 100 hash functions to choose from, even if our particular data set won't work well with Hash Function #48, that may be okay. Since we're going to pick a hash function randomly out of a hat, the odds of us picking Hash Function #48 for our hash table are only 1 out of 100.

So, with randomized hashing, when we instantiate a new hash table, the hash table will randomly pick the hash function it'll use. And once it decides on a hash function, it *must* use that hash function forever. If we hashed each key using a different hash function, we'd never be able to find those keys ever again unless we knew what hash function we used for each key. And that's certainly not something we want to keep track of.

When we write code that uses multiple hash tables, it's likely that each hash table is using a different hash function. But again, each hash table will stick with its unique hash function forever.

Hash Function Families

For a hash table to pick a random hash function, we need to first create a pool of potential hash functions to choose from. One way we *could* do this is to create a list of many of the different known hash functions out there. The Division Method is one viable hash function, but there are plenty of others, some of which have rather interesting names like MurmurHash, CityHash, FarmHash, and SpookyHash.

However, there's a simpler way to create a pool of hash functions. That is to create what is called a *hash function family*. A hash function family is a group of hash functions that all use the same general hashing method, except that they differ with regard to some other detail. This will make more sense with an example, so let's go ahead and create a hash function family out of division hashing.

Our approach will be to have lots of different hash functions that *all* use the Division Method, *but each hash function will divide values by different numbers*.

Let's go back to our earlier example where M was 89. If K is 412341439, we've learned that we'd compute `K % M` like so:

```
412341439 % 89 = 78
```

Now, we can't create other hash functions that have a different M. Remember, M corresponds to the size of our hash table's underlying array. If the underlying array has 89 slots, we're stuck dividing all of our values by 89.

However, we *can* modify our division formula slightly so that we're going to perform not one, but *two* modulo operations. To do this, we're going to choose a *second* prime number, which we'll call P, and run the following formula:

```
K % P % M
```

For our example, we'll say that P is 10061, which is a prime number. This gives us:

```
412341439 % 10061 % 89 = 80
```

Thus, the hash code comes out to be 80.

This new variable, P, becomes the key to creating a hash function family. Specifically, we can create multiple hash functions where each hash function uses a different value for P. While they must all use the same value for M, there's no reason why they can't have different Ps. Let's see how this allows us to create a hash function family and eventually solve our problem of division hashing working poorly for a particularly unlucky data set.

Continuing with our example of M being 89, let's create five different hash functions. Each hash function will have one of the following possible P's: `10037`, `10039`, `10061`, `10067`, or `10069`. When each of these hash functions hashes the same K of 412341439, we end up with five different hash codes:

```
412341439 % 10037 % 89 = 70
412341439 % 10039 % 89 = 69
412341439 % 10061 % 89 = 80
412341439 % 10067 % 89 = 66
412341439 % 10069 % 89 = 35
```

At the same time, since each hash function also divides the results by the same M of 89, we ensure that each hash code will all be within the range of 0 through 88. Again, this is exactly what we want if our hash table is to distribute values into indexes 0 through 88.

And so, we've successfully created an effective hash function family. In sum, this family consists of the following five hash functions:

```
Hash Function #1: K % 10037 % M
Hash Function #2: K % 10039 % M
Hash Function #3: K % 10061 % M
Hash Function #4: K % 10067 % M
Hash Function #5: K % 10069 % M
```

Naturally, we can create *hundreds* of hash functions along these lines. We need to find hundreds of prime numbers that can fill in for P. And indeed, those prime numbers exist; there are plenty of prime numbers to go around.

With all this in mind, it's pretty straightforward for the computer to select a hash function at *random*. All it needs to do is choose a random value for P. By picking a random P, we've effectively picked a random hash function. This is true even though it's already fixed that the hash function's general strategy will be to employ division hashing.

Let me bring this all back and spell out how we've solved our problem. Again, our concern was that there might be a data set out there that simply doesn't get distributed uniformly by our chosen method of hash function, such as division hashing, for example. Imagine, for example, that M was 89 and all the integers in our data set were divisible by 89. If we decide to use the Division Method for our hashing approach, all the integers would get shoved into index 0!

By having our hash table pick a P at random, there are high odds that the Division Method will indeed distribute a given data set uniformly. We'd only get messed over if the items in our data set were all divisible in the same way by both M *and* P. Besides being extremely unlikely, it can in any case no longer be said that there might be a data set that doesn't distribute well with the Division Method. The Division Method would indeed work well for almost any P that we end up selecting.

And that makes one big, happy, hash function family.

Code Implementation: Randomized Hash Functions

It turns out that it takes relatively minimal code to implement random hashing, at least at a basic level:

```
import random


class DivisionHasher:

    def __init__(self, array_length):
        self.array_length = array_length

        # Choose a random prime number:
        p = random.randint(1000, 10000)
        while not self.is_prime(p):
            p = random.randint(1000, 10000)

        self.prime = p

    def hash(self, key):
        return key % self.prime % self.array_length

    # Fermat's Primality Test
    def is_prime(self, number):
        for _ in range(100):
            a = random.randint(1, number - 1)
            if pow(a, number - 1, number) != 1:
                return False

        return True
```

Let's take a stroll through the code.

We've created a `DivisionHasher` class that acts as a "machine" that hashes a value (which in our code is referred to as `key`) into a hash code. The `DivisionHasher` class accepts an `array_length` variable that represents what we've been calling M—that is, the length of a hash table's underlying array.

To make things a little easier on us, we rely on the user to tell us what they want their hash table size to be.

The next bit of code implements the randomization part of random hashing. As we did earlier, our approach to random hashing is to pick a random prime P that will be used for the computation of `K % P % M`. This P effectively defines which hash function from our family we'll be using. Here, we randomly pick P and assign it to the variable `self.prime`:

```
p = random.randint(1000, 10000)
while not self.is_prime(p):
    p = random.randint(1000, 10000)

self.prime = p
```

Here's how we pick a random prime. We choose a random integer between `1000` and `10000`. For this basic proof-of-concept example, we're assuming that the integers being hashed are all greater than `1000`. For other types of integers, this range would have to be adjusted.

We then ensure that the integer is prime by continuously picking integer after integer until we find one that's prime. This uses a helper method called `is_prime`.

Now, here's the fun part: how does the `is_prime` method work? Well, it uses Fermat's Primality Test, which we looked at in the previous chapter! Cool beans.

The real action of our code, though, is the `hash` method. This method accepts a `key` and returns a hash code using the snippet:

```
return key % self.prime % self.array_length
```

This should look pretty familiar. It's our formula of `K % P % M`!

It should also be noted that this `hash` method only works on a `key` that's an integer. If you wanted to hash a string, for example, you'd first perform some computation to convert the string to an integer. You could, theoretically, do this by converting each character to its corresponding ASCII code and then multiplying or summing all the ASCII codes together. We'll explore this more in the next chapter.

I'll emphasize that this entire implementation is bare-bones, and there are *many* optimizations that can—and should—be made. My goal here, though, is to simply present code that conveys the main ideas of this chapter.

Code Implementation: Bare-Bones Hash Table

Now that we've gotten this far, we may as well have fun by building our own hash table from scratch. Again, you are *way* better off using Python's built-in dictionary, but this code implementation is here to help concretize the concepts you've learned so far.

We'll create our own hash table in two passes. First, we'll implement a hash table that is *extremely* basic but demonstrates how it makes use of the `DivisionHasher` class. Then, we'll add a couple more features (but the result will admittedly still be pretty bare-bones).

First Pass

Here's the first basic hash table implementation:

```
import division_hasher


class HashTable:

    def __init__(self, array_length):
        self.array = [None] * array_length
        self.array_length = array_length
        self.hasher = division_hasher.DivisionHasher(array_length)

    def insert(self, key, value):
```

```
        hashcode = self.hasher.hash(key)
        self.array[hashcode] = value

    def search(self, key):
        hashcode = self.hasher.hash(key)
        return self.array[hashcode]
```

Like the `DivisionHasher` class, this `HashTable` relies on the user to decide the size of the hash table's underlying array. This is the argument `array_length`.

Here's the constructor:

```
def __init__(self, array_length):
    self.array = [None] * array_length
    self.array_length = array_length
    self.hasher = division_hasher.DivisionHasher(array_length)
```

First, we create the underlying array, which we call `self.array`. We give it the size of `array_length` and fill each slot with `None` for now.

Next, we save `array_length` as the instance variable `self.array_length` since we'll need to access it later.

Then, we choose which hash function our hash table will use. Here, we're using the `DivisionHasher` class we implemented earlier, which represents the division hash function family. However, we could easily swap this out for some other hash function family as long as we use a class that implements a `hash` function. In any case, our hash function gets stored in a variable called `self.hasher`.

Next, we have the `insert` method. We allow a user to insert a key-value pair into the hash table using code like this:

```
hash_table = HashTable(89)  # array's size is 89
hash_table.insert(55, 17)
```

That is, we'll insert a key of `55` which has a corresponding value of `17`. Although throughout our discussion in this chapter, we've assumed that the key and value are identical, in real life, the key and value are usually different, as is the case here.

Here, again, is the `insert` method:

```
def insert(self, key, value):
    hashcode = self.hasher.hash(key)
    self.array[hashcode] = value
```

First, we use `self.hasher` to hash the `key` into a `hashcode`.

Then, we place the `value` into the `self.array` at the index which is the `hashcode`. So, if the `hashcode` for `55` is `9`, then the `value` of `17` will be placed at index `9`.

We also implement a `search` method, which allows a user to look up a value by its key, such as:

```
hash_table.search(55)
```

This will return `17`.

In this first pass, the code for the `search` method is short and is similar to the `insert` method:

```
def search(self, key):
    hashcode = self.hasher.hash(key)
    return self.array[hashcode]
```

Here, we hash the `key` and get a `hashcode`. This `hashcode` represents the index in `self.array` where our desired value will be found, so we go find it there.

Second Pass

Here's a slightly better `HashTable`, but again, I'll emphasize that it's still not nearly as robust as the real deal.

This version adds two important features. One is a `delete` method since it's pretty common to delete keys from a hash table. The other is that we now handle *collisions* of keys, a problem I discussed in Volume 1, Chapter 8. In short, the issue of collisions is that it's possible for two different keys to be hashed into the same hash code. This means that we have to somehow fit both values into one array slot.

One approach for handling this is called *separate chaining*, which stores multiple values in each array slot using another array (or linked list), which I'll call a "subarray."

Here's the specific way we'll do this. If our hash table has five slots, each slot will itself start out holding an empty array, like so:

```
[
  [],
  [],
  [],
  [],
  []
]
```

Furthermore, we'll need to store not just the values but also the *keys* so that we can identify which value goes to which key.

So, let's say that we insert the following key-value pairs:

```
Key: 3, Value: "a"
Key: 9, Value: "b"
```

If both keys `3` and `9` hash into the same hash code, say `0`, here's what the hash table's underlying array will look like:

```
[
   [
      [3, "a"],
      [9, "b"]
   ],
```

```
    [],
    [],
    [],
    []
]
```

That is, in each subarray we'll add a key-value pair in the form of yet another array (a sub-subarray, I guess), where index `0` is the key and index `1` is the value.

Here's the code:

```
import division_hasher


class HashTable:

    def __init__(self, array_length):
        self.array_length = array_length
        self.array = [[]] * self.array_length
        self.hasher = division_hasher.DivisionHasher(self.array_length)

    def insert(self, key, value):
        hashcode = self.hasher.hash(key)

        for key_value_pair in self.array[hashcode]:
            if key_value_pair[0] == key:
                key_value_pair[1] = value
                return

        self.array[hashcode].append([key, value])

    def search(self, key):
        hashcode = self.hasher.hash(key)

        for key_value_pair in self.array[hashcode]:
            if key_value_pair[0] == key:
                return key_value_pair[1]

        return None

    def delete(self, key):
```

```
        hashcode = self.hasher.hash(key)

        for index, key_value_pair in enumerate(self.array[hashcode]):
            if key_value_pair[0] == key:
                del self.array[hashcode][index]
```

As you can see, a significant change in this version is that in the constructor, instead of filling each slot of `self.array` with `None`, we fill it with a blank subarray. This way, we can hold multiple key-value pairs in each slot of `self.array`.

This has ramifications for all the methods of our class. In the `insert` method, we now don't simply insert a value but append to the subarray another array that contains the key and value. Hence, the code: `self.array[hashcode].append([key, value])`.

To improve the `insert` method further, I also provided the ability to overwrite a key and give it a new value:

```
    for key_value_pair in self.array[hashcode]:
        if key_value_pair[0] == key:
            key_value_pair[1] = value
            return
```

For example, we may decide that the value associated with the key `3` should now be `"z"` instead of `"a"`.

Separate chaining also changes the way we search for values. In the updated `search` method, you'll note the newly added code that peers inside the proper slot of `self.array` and performs a linear search to find the correct key-value pair.

Lastly, I added a `delete` method. For example, if we want to delete the key of `3`, we'd call the `delete` method like this:

```
    hash_table.delete(3)
```

The `delete` method hashes the key into a `hashcode` and then searches for the key inside the appropriate slot of `self.array`. If we find the key, we delete the entire key-value pair from `self.array`.

And so, we've created our own hash table from scratch. While it's not nearly as good as a Python dictionary, and you don't want to use it in real life, going through the process can help you better understand how hash tables work under the hood.

Python Dictionaries' Hash Function

I mentioned that there are many different types of hash functions out there. The Division Method is one of the simplest approaches, but there are plenty of more complex and nuanced hash functions in the wild.

At one point, Python's dictionary class used a hash function called Fowler–Noll–Vo, or FNV for short. However, Python version 3.4 switched things up and now uses another hash function called SipHash. Indeed, SipHash uses randomized hashing and hash function families. In the words of the SipHash documentation:

"SipHash is a family of pseudorandom functions (aka keyed hash functions) optimized for speed on short messages."

For a fascinating read as to why this switch was made, you can find the details at https://peps.python.org/pep-0456. It also makes for great dinner conversation.

Wrapping Up

We took a deep dive into hashing in this chapter. You learned about division hashing, hash function families, and how picking a hash function at random can help avert a worst-case scenario for a hash table.

In the next chapter, we're going to take a look at a clever Monte Carlo algorithm that uses hashing to solve a common problem called *substring search*. In fact, there's a good chance that you rely on substring search daily without even thinking about it. Along the way, we'll also discuss some crucial computer science concepts, such as base number systems and how binary numbers truly work.

Onward!

Exercises

The following exercises provide you with the opportunity to practice with hash functions, randomized hashing, and hash function families. The solutions to these exercises are found in the section *Chapter 10*.

1. Imagine that you're the nefarious hacker I described in the chapter. You've found an app to exploit, and you obtained its source code. (Bwah hah hah hah!) When reading the code, you discover that it stores its data in a hash table, and the hash function being used is the Division Method. However, it's not randomizing the hash function in any way. Instead, the app always uses the following hash function, with `K` representing the key being hashed:

   ```
   K % 997
   ```

   What data can you feed the app so that all the data ends up in the same slot within the hash table?

2. *Exploration:* In this chapter, we only hashed numbers, but what if we want to hash strings? Let's extend our division hashing method so that it can hash strings as well. To do this, we'll rely on ASCII standards to map each alphabet character to an integer. In Python, we can use the `ord` method to convert a character to an integer. For example:

   ```
   ord('a')
   >>> 97

   ord('z')
   >>> 122
   ```

   Modify the `hash` function from our `DivisionHasher` class so that it can hash strings as well as integers.

3. *Exploration:* Once you've completed Exercise #2, here's another thing to think about. Does your hash function place anagrams in the same slot? For example, does your hash function assign the same hash code to both `"listen"` and `"silent"`? Can you make it so that your function doesn't necessarily assign the same hash code to anagrams?

Chapter 11

Keeping Your Text Search Sharp with a Little Rabin-Karp

In the previous chapter, you gained deeper insight into hash functions and even built your own basic hash table using division hashing. However, hash functions can be used even outside the context of hash functions to solve all sorts of different problems, sometimes in surprising ways.

In this chapter, you'll discover one innovative way in which hash functions, when wrapped in a Monte Carlo algorithm, can be used to solve a common and fundamental problem known as *substring search*.

Substring Search

When you're working on your code in your text editor and you need to find the variable `doris_the_cat` somewhere within thousands of lines of code, it's unlikely you'll go scrolling down the code until the variable catches your eye. Instead, you'll probably use your text editor's "find" feature, in which you'll enter `doris_the_cat`, or just `doris`, and your editor will immediately put the first instance of that variable in view. While most of us take this feature for granted, it's not at all trivial to find the string `doris_the_cat` amid thousands of lines of code so quickly.

This problem is known as *substring search*. That is, we have some large body of text (such as a code file), and a smaller string (such as `doris_the_cat`), and we need to find the smaller string within the larger body of text. We can think of this as searching for a needle within a haystack. In computer science jargon, the needle is called the *pattern*, and the haystack is called the *text*. However, I'll continue to use the terms "needle" and "haystack," since the term "text" is vague and can make things confusing.

Numerous algorithms exist that solve substring search, and researchers are still on the lookout for even faster alternatives. We'll look at a couple of them in this chapter, starting with brute force.

Brute-Force Substring Search

The idea behind brute-force substring search is to use two loops to search the haystack. The outer loop combs through each character of the haystack one at a time, and when it finds a character that matches the first character of the needle, an inner loop begins. Before we get to the inner loop, though, let me show you what I'm talking about.

Say that our needle is the string `"bet"` and the haystack is the string `"flibbertigibbet"`. (Of *course* that's a real word. A flibbertigibbet is an excessively talkative person.) We begin with two pointers, one that points to the *needle's* first character, and the other that points to the *haystack's* first character:

```
↓
bet

↓
flibbertigibbet
```

We check if the two characters match. Currently, they don't, so we move the haystack's pointer onward:

```
↓
bet

 ↓
flibbertigibbet
```

Again, the haystack's letter `"l"` does not match the needle's letter `"b"`, so the haystack pointer needs to move on. Let's skip to the exciting part, where the haystack and needle pointers *do* match:

Oh, a match! Both pointers point to `"b"`s. Now, this is when we begin the inner loop I mentioned earlier. That is, we keep the haystack pointer in place, but set a new, *second* haystack pointer—represented by an emoji in the diagram shown—to scan the rest of the needle. I'll call this second pointer the "emoji pointer." (I want to give this pointer a unique name, so there it is.)

After initializing the emoji pointer, we move the needle pointer along to the next character. We then check whether the needle pointer and the emoji pointer point to matching letters.

The emoji pointer is sad because the `"b"` that it points to does not match the `"e"` of the needle pointer. This means that we have not found our matching substring—yet.

To continue our search, we terminate our inner loop and get rid of the emoji pointer for the time being. We reset the needle pointer to point back to the needle's first character and begin the next round of the outer loop. As we've done with the previous rounds of the outer loop, we next compare the haystack and needle pointers to see if we have a match:

And sure enough, we do! This means we begin an inner loop once again, which also means we get to bring our emoji pointer back, and also increment our needle pointer. We then check to see whether the emoji pointer and needle pointer yield a match:

This is terribly exciting because the emoji pointer and needle pointer both point to an `"e"`! We now have matching substrings of `"be"`. Are we about to find our needle of `"bet"`? Well, let's move the emoji and needle pointers along and see:

Bummer, it's not a match. Our inner loop terminates, and we've got to reset our needle pointer back to the beginning, dispense with the emoji pointer, and move on. We'll skip a few steps once again and reach the climax of our search:

The `"b"`s match, so we fire up an inner loop. We activate our emoji pointer and move the needle pointer along:

Again, a match! Will we, once and for all, find our complete needle?

You bet! At this point, our algorithm will return the haystack index where the beginning of the needle can be found.

And that wraps up our brute-force substring search walkthrough.

Code Implementation: Brute-Force Substring Search

Here's one way to implement brute-force substring search:

```
def find_needle(haystack, needle):
    for haystack_index in range(len(haystack) - len(needle) + 1):
        for needle_index in range(len(needle)):
            if needle[needle_index] != haystack[haystack_index +
needle_index]:
                break

            if needle_index == len(needle) - 1:
                return haystack_index

    return None
```

Here, the `haystack_index` serves as the haystack pointer and `needle_index` as the needle pointer. We get our "emoji pointer" by adding the `haystack_index` and `needle_index` together, rather than declaring an explicit "emoji pointer" variable (which would be weird).

Once the `needle_index` reaches the end of the needle, it means we found the needle in the haystack. Accordingly, we return the `haystack_index`, which is the location of the needle's first character within the haystack.

The Efficiency of Brute-Force Substring Search

In many practical cases, brute-force substring search isn't half bad in terms of speed. In looking at the previous example, our haystack pointer touched each haystack character once, which is O(N) if N represents the number of characters in the haystack.

For clarity's sake, I'm going to use the variable *H*, instead of N, to refer to the number of haystack characters, with H standing for "haystack." I will instead use N to refer to the number of characters in the "needle," since "needle" starts with the letter N. In short, H is the number of haystack characters, and N is the number of needle characters. Got it?

So far, then, we can say that our haystack pointer will point to all H characters of the haystack.

The additional steps of the algorithm consist of moving the needle and emoji pointers, which we do in tandem within an inner loop. In the earlier example, we only launched this inner loop a handful of times. We also saw how the inner loop terminated after a couple of iterations before hitting a mismatch between the needle and emoji pointers. In fact, the inner loop will never run more times than the number of N needle characters. That is, if the loop does manage to run N times, that means we'll have scanned the entire needle, and there's nothing left for the loop to do. By that point, we'll know whether the needle was found at that point in the haystack or not.

In the earlier walkthrough, only a handful of inner loop steps were executed. We might be tempted to simply discount those steps entirely and say that our algorithm ran in approximately O(H) time. Indeed, brute-force substring search does take O(H) time for average-case scenarios. However,

as you know, when evaluating an algorithm's efficiency, we also need to take into account the *worst*-case scenario.

Here's an example of a kind of worst-case scenario for substring search. Say that the needle is `"aaaab"` and the haystack is `"aaaaaaaaaaaaaaaaaaab"`. In this case, for every haystack character (save for the last few), we have to activate our inner loop. And the inner loop itself will scan the full length of the needle. Therefore, we'd say that brute-force substring search can take up to O(HN) in the worst case. This can be slow if we're dealing with a lot of text.

Luckily, there are a number of algorithms out there that improve the performance of substring search. These are perhaps the three most famous:

- Knuth-Morris-Pratt (otherwise known as KMP)
- Boyer-Moore
- Rabin-Karp

All three of these algorithms run in O(H+N) time even for the worst-case scenario. Note that O(H+N) is *much* faster than O(HN).

Of these three algorithms, we'll explore the Rabin-Karp algorithm. It ties in nicely with many of the concepts discussed earlier in this book, and it will also unlock a bunch of new and useful concepts. One of those new and useful concepts is a technique that many refer to as the "sliding window" technique. So, let's kick things off with that.

The Sliding Window Technique

Here's a seemingly simple problem that has nothing to do with substring search. Say we have the array `[3, 2, 7, 4, 6, 3, 5, 8]`, and we want to find the greatest contiguous four integers that yield the greatest sum when added together. Here's what I mean.

The first four integers of the array `3, 2, 7, 4` are contiguous, meaning they're all in a row. And when we add them together, we get a sum of 16. Cool.

The problem I'm proposing is to find the set of four contiguous integers that will yield the *greatest* sum in the array. For example, there's another set of four contiguous numbers—`7, 4, 6, 3`—that add up to 20. Is this the greatest sum we can get, though? Let's devise an algorithm to figure this out.

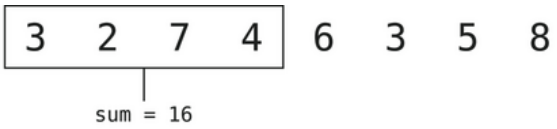
One approach we can take is brute force. Specifically, we simply try out every set of four contiguous numbers and keep track of which set gives us the greatest sum. That is, we try `3, 2, 7, 4` and then `2, 7, 4, 6` and then `7, 4, 6, 3`, and so on.

To articulate the time complexity of this approach, we need to note that we have two variables to contend with. First, we have the length of the array, which we'll call N. But we *also* have the number of how many contiguous integers we're summing up. In our example, we're working with sets of 4 contiguous numbers, but alternative problems might have us find the greatest sum of 3 numbers or 5 numbers. We'll call this second variable K.

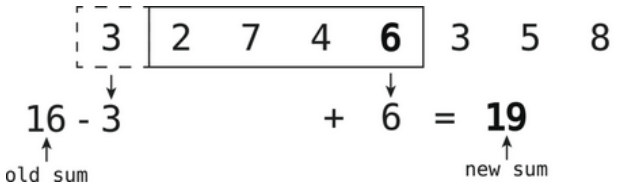
It turns out that for the brute-force approach, for each of the N elements of the array, we have to compute the sum of K elements. The only saving grace is that we don't need to do this for the last three elements. Once we calculate the sum of the final four elements, we're done—since there are no more groups of four elements after that point.

In the end, brute force here takes about N*K steps, which in Big O is expressed as O(NK). This has the potential to get unwieldy if our problem involved more numbers. If, for example, our array had 1,000 values and we were adding up 10 contiguous numbers, we'd have to perform 10,000 steps. However, we can use a more clever approach—the *sliding window technique*—to complete our task in O(N) time. Here's how it goes.

We begin by computing the sum of the first four integers:



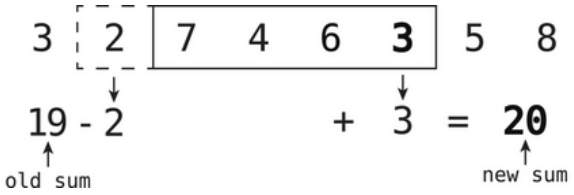
The box surrounding the four integers is our "window." So far, we haven't done anything clever. But watch what we do in the next step:



Instead of performing a brand-new computation to add up the next four integers, **2, 7, 4, 6**, we only perform one subtraction and one addition. That is, because we know that the *current* window shares the numbers **7, 4, 6** with the *previous* window, *we don't need to add those numbers up again.* The only difference between the current window and the old window is that the current window drops the **3** and adds an additional **6**. So, to compute the sum of the current window, we take the sum of the old window, and simply subtract **3** and add **6**.

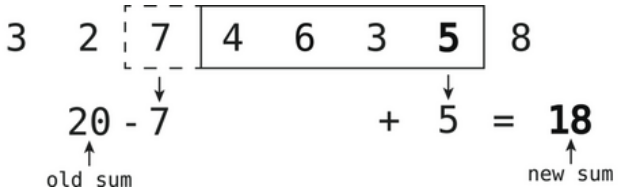
This is the essence of the sliding window technique. Because the window "slides" incrementally, we only need to compute the *differences* between the new window and the old window instead of recomputing *everything* over again.

Moving on with our walkthrough, the 19 yielded by the current window is greater than the 16 produced by the previous window. (We can use a variable to keep track of the greatest sum we've encountered so far.) Let's see what happens when we "slide the window" in the next step as shown in the diagram.

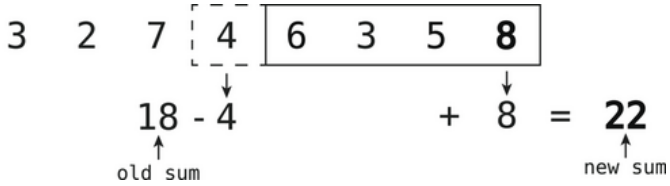


Here, we subtract 2 and add 3, giving us 20, which is the greatest sum encountered so far. Although you may grasp the idea by now, let's see this example to the end (even at the risk of you thinking me a flibbertigibbet).

We slide the window again:



This window yields 18. As of this point, 20 is still the greatest sum. We have one final step:



Aha! This final window sums to 22. Given that this is the greatest sum we've found, this is the result that our algorithm will output.

The efficiency of this approach is O(N) since we make a single pass through all N values. For each window, we perform a constant number of computations, that is, one addition and one subtraction. Even if the problem was changed so that we were searching for the greatest sum of 10—or even *100*—contiguous numbers, this doesn't affect the speed of our algorithm

whatsoever. No matter what, we'll only perform one addition and one subtraction for each of the N values. The K variable no longer matters.

This is a massive win since the sliding window algorithm executes in O(N) time vs. the O(NK) time of the brute-force approach.

Code Implementation: The Sliding Window Technique

Here is a Python implementation of the sliding window algorithm:

```
def max_sum_of_four_integers(array):
    current_window_sum = 0

    for i in range(4):
        current_window_sum += array[i]

    max_sum_so_far = current_window_sum

    for i in range(4, len(array)):
        current_window_sum += array[i]
        current_window_sum -= array[i - 4]

        max_sum_so_far = max(max_sum_so_far, current_window_sum)

    return max_sum_so_far
```

We begin by creating the initial window and storing the sum in the variable `current_window_sum`. We also create a variable `max_sum_so_far`, which will track the greatest window sum we encounter. We return this variable at the end of the function. To start, though, this variable will contain the sum of the current window.

We then begin a loop that "slides" the window along. We do this by starting the loop's index (`i`) at `4`. This is because `array[4]` is the value just to the right of the previous window. Because this is the new value we're adding to the new window, we add this value (`array[i]`) to `current_window_sum`.

At the same time, we subtract from `current_window_sum` the value at `array[i - 4]`, as this is the first value of the *previous* window. If the `current_window_sum` is greater than the `max_sum_so_far`, we update the `max_sum_so_far` to now be the `current_window_sum`. We then continue to slide the window by incrementing `i`, and repeat this entire process until we reach the end of the input `array`.

We're now ready for the Rabin-Karp algorithm, which, in fact, uses the sliding window technique.

Rabin-Karp Substring Search

The Rabin-Karp substring search algorithm is doubly clever, for it relies not on one, but *two* clever tricks to perform substring search in O(N) time.

The first trick is that it performs a *hash function* to convert the needle and haystack window into *integer hash codes*. That is, the entire needle becomes a single integer, and so does the haystack window. If these two integers are the same, it means that the needle and haystack window are the same.

The reason this is a big deal is that, as I noted earlier, it takes multiple steps to compare our needle to a section of the haystack. That is, comparing `"aaab"` to `"aaaa"` or `"ddde"` to `"dddd"` requires us to perform up to four comparisons. But when it comes to comparing two *integers*, well, a computer can do that in a single step, assuming that the integers aren't terribly long. Even if the integers have multiple digits, computers compare integers in constant time, something they cannot do with strings.

For example, say we use a hash function to first convert `"ddde"` into the integer `3334` and `"dddd"` into the integer `3333`. At this point, the computer only has to perform a single-step comparison of integers `3334` with `3333` to know that there's no match.

Now, the potential flaw with this first trick is that although comparing integers takes constant time, it takes multiple steps to hash all the characters of a string into an integer. This is because a hash function needs to perform a computation with *each and every character* of the string to produce a hash code.

For example, let's say our hash function works like this: We convert each character with a number according to the scheme that `"a"` = `0`, `"b"` = `1`, `"c"` = `2`, `"d"` = `3`, and so on. So, `"cab"` would become `201`, and `"bad"` would become

`103`. However, this still requires the hash function to process each letter to convert it into its corresponding numerical digit. If a string has 100 characters, this would take 100 steps. So, while the first trick is nice in theory, it would seem that it doesn't save us any time.

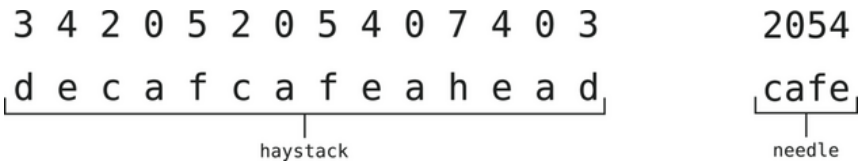
But this is where the second trick comes in. The second trick is that we use —you guessed it—the sliding window technique so that we never have to hash the same character more than once.

Now that you've seen the two "tricks" we'll be using, let's get into the fine details.

Rabin-Karp in Action

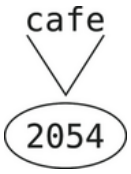
First, we'll look at a simplistic version of the Rabin-Karp algorithm, and from there, we'll move on to a more sophisticated approach. In this simplistic model, we're going to work with text that only contains the letters `a` through `j`. That is, we're going to deal with an alphabet that contains only 10 letters. We'll use the same scheme we did earlier, where `a` is `0`, `b` is `1`, all the way up to `j`, which is `9`. (Later, we'll extend this to the full 26-character English alphabet.)

Now, say we want to find the needle `"cafe"` within the haystack `"decafcafeahead"`. In the following diagrams, I put each letter's corresponding number right above it for ease of reference.



We're now ready to launch the Rabin-Karp substring search algorithm.

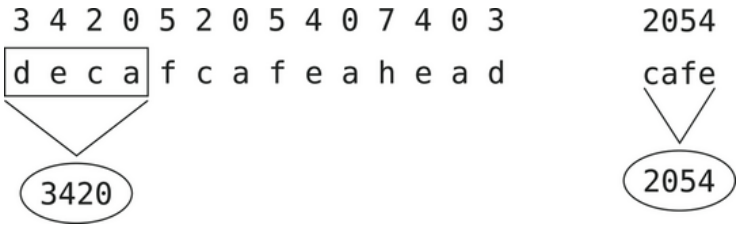
The first thing the algorithm does is perform our hash function on the needle `"cafe"`, turning it into a hash code of `2054` as shown in the diagram.



Now, it may seem straightforward to perform this conversion; after all, we're simply mapping each letter to its corresponding number. However, there's a tad more math going on here than first meets the eye. But we're going to defer that discussion until the next section. Let's get a simple overview of Rabin-Karp first.

At this point, all we've done so far is hash our needle. Going forward, Rabin-Karp performs the following steps:

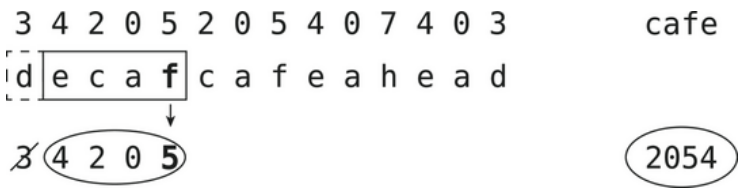
Step 1: We establish a "window" at the beginning of the haystack. Throughout the algorithm, *the haystack window will always be the same length as the needle*. In our example, the window spans four letters. We next hash this haystack window, and compare its hash code to the needle's hash code:



Because the haystack window hash code is `3420` and is most certainly not equal to the needle's hash code of `2054`, we know that our haystack window and needle do not match. The next step is where the sliding window technique kicks in. Going forward, I'm going to refer to the haystack window simply as the "window."

Step 2: We slide the window one letter to the right, encompassing the letters `"ecaf"`. Now, we could hash this new, second window of `"ecaf"` in the same way we did our first window of `"deca"`, but this would take too much time. Instead, we perform our sliding window trick, relying on the fact that the windows `"deca"` and `"ecaf"` share the same letters `"eca"` (which hashes to `420`).

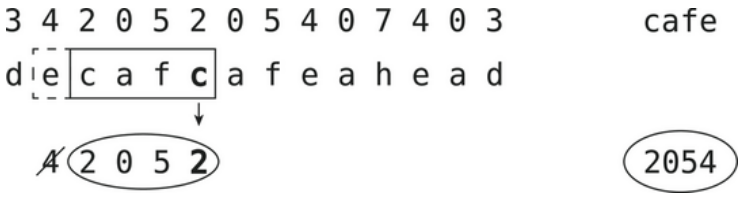
The only differences are that the first window had an extra `"d"` at the beginning and the second window has an extra `"f"` at the end. So, we do this:



That is, we drop the initial `3` and tack on a new `5` at the end. Brilliant, right? The `420` part doesn't have to change at all since both windows have that in common. This saves us from hashing the `420` digits again; we've already done that when we processed the first window. (Again, I'm glossing over some of the math for the moment.)

At this point, our current window is `"ecaf"`, and our sliding window hash function computed its hash code of `4205`. Because our needle's hash code is `2054`, this means we have not found our match.

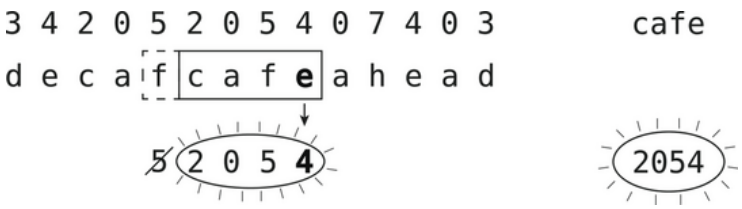
Step 3: The next step is to slide the window another notch to the right:



Applying the same sliding-window-hashing technique described earlier, we drop the old `4` from the beginning and tack on a `2` at the end, giving us a new hash code of `2052`. While this isn't far off from our needle's hash code of `2054`, it's not on the nose, so we have yet to find a match.

To move things along, we'll skip Steps 4 and 5 and get to the punchline.

Step 6: After sliding the window three more notches, we get this:



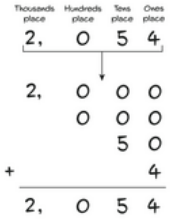
Here, the current window computes a hash code of `2054`, which is an exact match of our needle's hash code. This means we found a match! Indeed, both the window and our needle are the same characters, `"cafe"`.

The Rabin-Karp Hash Function

The hash function described in the previous section seems pretty simple. We take a string, such as `"bcd"`, and convert it into an integer based on the number that each character corresponds to. Because `b` is `1`, `c` is `2`, and `d` is `3`, we convert `"bcd"` into `123`.

This hash function would be as simple as I described it if we were converting the string `"bcd"` into a *string* that was `"123"`. But we're not. We're converting the string `"bcd"` into the *integer* `123`. As noted earlier, the reason why we make the hash code an integer is that it's faster to compare two integer hash codes than it is to compare two string hash codes. The thing is, though, that it takes a little math to convert a string into an integer.

To understand why this is so, you need to close your eyes and take a trip down memory lane. Picture yourself as a student in the third-grade classroom. At the front of the classroom, Mrs. Wilson was up at the chalkboard (remember those?) and teaching math. She was pontificating about the true meaning behind multidigit numbers, that is, numbers that contain more than one digit. On the chalkboard, she demonstrated that the meaning behind the number 2,054 is actually:



That is, the `2` in `2054` represents the thousands place, the `0` the hundreds place, the `5` the tens place, and the `4` the ones place.

Okay, you can open your eyes now.

Based on what Mrs. Wilson taught us, if we want to convert `"cafe"` to `2054`, our hash function cannot simply convert the `c` from `"cafe"` into a `2`. *It needs to convert the `c` into a `2000`.*

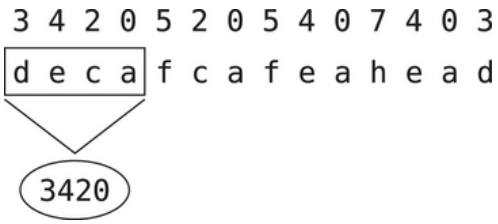
So, to convert the entire string `"cafe"`, we need our hash function to do the following:

a=0	c=2	e=4	f=5

```
c ⟶ 2 | x 1,000 = 2,000
a ⟶ 0 | x   100 =     0
f ⟶ 5 | x    10 =    50
e ⟶ 4 | x     1 =     4
                  2,054
```

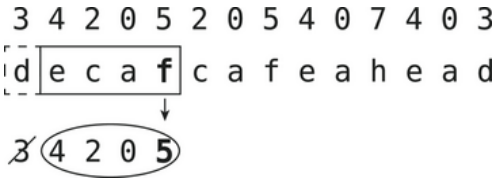
This isn't terribly complicated, and we'll look at the code for this in a little bit. In any case, this is the hash function we'll use to hash our needle and the first window from our haystack. Now, let's move on to how our hash function should work as we slide our window through the haystack.

Let's back up to the beginning of our example. We started off by hashing the first window of `"deca"` into `3420`:



So far, so good. We just saw how this hash function works.

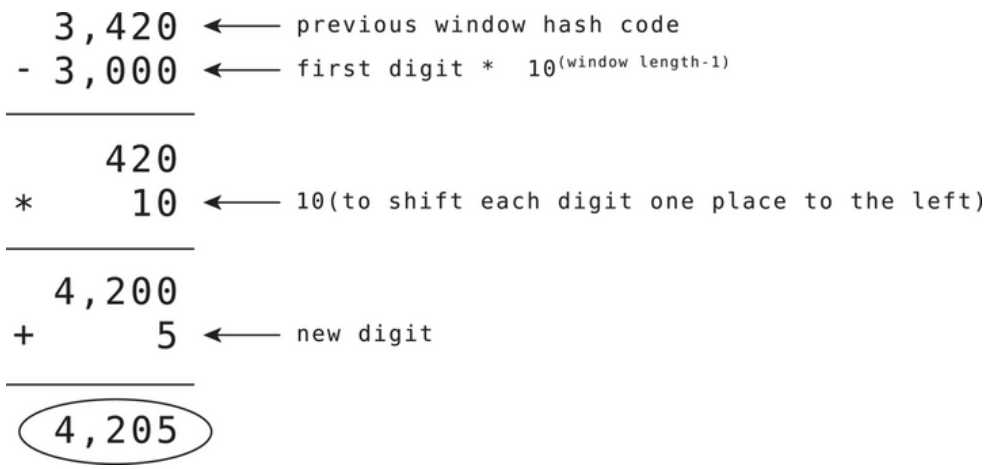
In the next step, though, we did this:



Now, what the hash function does here is not identical to hashing the first window. Because we're using the sliding window technique, we don't bother to hash `"ecaf"` from scratch. Instead, we eliminate the starting `3` from the previous hash code, and the only new item we need to hash is the final `"f"`. This becomes a `5`, which we tack on at the end of `420`, turning `420` into `4205`. So how does this hash function work, exactly?

The thing here is to remember Mrs. Wilson. We're dealing with multidigit numbers, so we need to reckon with our thousands place, hundreds place, and so on.

The sliding window hash function goes like this. (I'll show you a visual depiction of the computation first, and then explain it.) Here's how we convert `3420` into `4205`:



Let's break this calculation down. Again, our goal is to do two things to the `3420` in order to convert it into `4205`: we need to drop the initial `3`, and we need to tack on a `5` at the end.

First, we drop the starting `3`. Because that `3` represents `3000`, we need to subtract `3000` from `3420` to eliminate that `3`. This leaves us with a remainder of `420`.

Now, you and I know by looking at the `3420` that its `3` represents `3000`. But a computer didn't take Mrs. Wilson's class, so somehow, we need to tell the computer how to convert that `3` into `3000`. In other words, how do we get the computer to understand that the `3` represents the thousands place?

The key is understanding that what the first digit represents is tied directly to the haystack window size. When the window size is 4, the first digit will represent the thousands place. But if our window size was 3, then the first digit is the hundreds place, and if the window size was 5, then the first digit would represent the ten-thousands place.

So, to figure out what number to multiply the first digit by, we take `10` and raise it to the power of the *window size minus 1*. In our example, the window size, 4, minus 1 is 3. When we raise 10 to the power of 3, we get 1,000. We can then multiply this number by our first digit to see what the first digit represents. In our case, where the first digit is `3`, this gives us `3000`.

If our window size were 5, then we'd raise 10 to the power of 4 and get 10,000. If our first digit is `3`, we'd know that the `3` represents `30000`.

In any case, in our example, we've computed that the first digit represents `3000`. And so, when we subtract that from `3420`, we end up with `420`, which means that we successfully dropped the initial `3`. Our next step is to convert the `420` into `4205`.

Thankfully, this part is a little more straightforward. We convert the `420` into `4200` by multiplying it by `10`. In effect, this moves each digit one notch to the left to make room for our new digit. Once we have this result of `4200` in hand, we simply add the new, final digit of our current window to it. Since in our example the final digit is `5`, this gives us `4205`.

By the way, a hash function that employs the sliding window technique is known as a *rolling hash function*.

Note that our hash function for hashing the first window (as well as the needle) is a plain old regular hash function. I'll refer to this as the "initial hash function." But the hash function we use for hashing the second and subsequent windows is the "rolling hash function" I just described.

Whew! Now that we've seen how our hash function works, let's write up some code.

Code Implementation Rabin-Karp for Base 10

Our current illustration of the Rabin-Karp algorithm is still a simplistic variant, but we're not far off from the full algorithm. However, let's spin up some code for what we've covered so far:

```
def find_needle(haystack, needle):
    needle_hash_code = initial_hash(needle)
    window_hash_code = initial_hash(haystack[0:(len(needle))])
```

```
    if needle_hash_code == window_hash_code:
        return 0

    for index in range(1, len(haystack) - len(needle) + 1):
        drop_character = haystack[index - 1]
        new_character = haystack[index - 1 + len(needle)]
        window_hash_code = rolling_hash(window_hash_code, len(needle),
                                        drop_character, new_character)

        if needle_hash_code == window_hash_code:
            return index

    return None


def initial_hash(string):
    power = 0
    result = 0

    for char in reversed(string):
        result += character_hash_code(char) * 10**power
        power += 1

    return result


def rolling_hash(hash_code, window_length, drop_character, new_character):
    drop_number = \
        character_hash_code(drop_character) * 10**(window_length - 1)
    result = hash_code - drop_number
    result *= 10
    result += character_hash_code(new_character)

    return result


def character_hash_code(char):
    return ord(char) - 97
```

Here, our primary method is `find_needle`, which accepts both the `haystack` and `needle` arguments. If the `needle` is found, the method will return the index of

where the `needle`'s first character is found within the `haystack`. If the needle is nowhere to be found, we return `None`.

The `find_needle` method relies on the two hash functions I described earlier, namely, our initial hash function and our rolling hash function. I placed the code for these hash functions in their own distinct methods: `initial_hash` and `rolling_hash`, respectively.

First, the `find_needle` method performs the `initial_hash` function on the `needle` and the first window of the `haystack`. This provides us with the hash codes of both the `needle` and the first `haystack` window. At this point, we check whether the hash codes are the same, which would indicate that the `needle` and `haystack` window are both the same. If they are, we return `0` to indicate that the `needle` can be found at index `0` within the haystack.

If we don't have a match, we then begin a loop that iterates over almost every index of the haystack. We begin at index `1`, and end at whichever index will be the first character of the final window. Within this loop, we identify the `drop_character`, which is the first character of the old window. Likewise, we locate the `new_character` that is being introduced as the final character of the *new* window.

We compute the new window's hash code by performing the `rolling_hash` function, and then check to see if the new window's hash code matches the `needle`'s hash code. If we find a match at any point, we return the current `index`, which is where our current haystack window begins. If we get through the entire `haystack` without finding the `needle`, we simply return `None`.

The `initial_hash` and `rolling_hash` functions track with the math I described earlier. Note that they both rely on yet another function, `character_hash_code`, which returns the hash code for a single character. I've set it up so that `"a"`

will return `0`, `"b"` will return `1`, and so on. To do this, I subtract `97` from whatever Python's built-in `ord` function returns since `ord("a")` is `97`, and `ord("b")` is `98`, and so on, corresponding to the characters' ASCII codes.

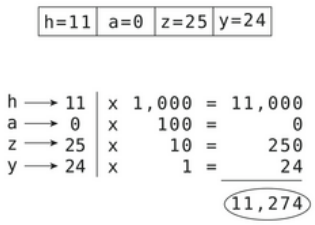
Collision Ahead

Although we've seen the gist of the Rabin-Karp algorithm, we need to iron out a few more details. The first item is that, until now, we've dealt with examples where our strings only contain letters `a` through `j`. Ultimately, though, we want our algorithm to work with the full 26-letter English alphabet. In fact, we want it to work for even larger character sets that include lowercase and uppercase letters, numbers, punctuation marks, and more. Indeed, the full ASCII character set contains 256 possible characters. But for now, let's deal with the set of 26 lowercase alphabet letters.

Here's why we care about the size of the character set.

Let's extend our letter-to-number scheme to the rest of the alphabet, even beyond the letter `j`. So `h` corresponds to `10`, `i` corresponds to `11`, and so on, until `z`, which corresponds to `25`. (The letter `a` corresponded to `0`, so our final letter, the `z`, ends at `25`.)

Here's what happens if we try to hash the word `"hazy"`:



While this computation seemed to work out, there's a subtle flaw here.

Let's zero in on the `"z"` and the `"y"`. In fact, let's pretend that the entire string was just `"zy"`. If we hash `"zy"` according to this method, we get `274`.

The problem is that `"zy"` *isn't the only string* that has a hash code of `274`. After all, `"che"` *also* hashes to `274`.

This was *not* a problem when we only dealt with letters `a` through `j`, though. Because there are only 10 possible characters, we are guaranteed that each character's corresponding digit (`0` through `9`) will only take up one "place" in the larger number. That is, if the final character of our string is `"e"`, its corresponding number of `4` will be entirely contained within the ones place. Even the highest available character—`"j"`—corresponds to `9`, which is a single digit.

But if our final letter is `"k"`, which corresponds to `10`, that's a double-digit number, so the hash code "bleeds" from the ones place into the tens place. This, in turn, introduces the problem we saw earlier. That is, the hash code `10` *could* represent `"k"`, but it could *also* represent `"ba"` since `"b"` is `1` and `"a"` is

`0`. There's no longer a way to know for certain what string a given hash code is supposed to represent.

In fact, you can even have two different strings of the same length that convert to the same hash code. For example, both `"lk"` and `"ma"` have the hash code `120`. This is because `"m"` is equivalent to `12`, and `"a"` is `0`. Because the `a` converts to a single digit, it doesn't bleed at all into the tens place. This gives us a hash code of `120` since we have `12` tens and `0` ones.

But the string `"lk"` is an entirely different story. The character `"l"` converts to `11`, and `"k"` converts to `10`, which is *also two digits*. This `10` bleeds from the ones place into the tens place, and also causes `"lk"` to have a hash code of `120`. That is, we have `11` tens, and `10` ones. This is `110 + 10 = 120`.

Our algorithm might mistakenly think it found a match if our needle is `"lk"` and the haystack window is `"ma"`! Because of this, our hash function needs to make sure that when we convert a string into a numerical hash code, each character of that string converts into a number that can be contained *within a single digit place*.

So, how do we apply Rabin-Karp to a character set that contains more than ten letters?

Covering All Our Bases

The solution to this problem is to use a different *number base* to represent our numbers. While our day-to-day number system uses base 10, if we want our hash function to cover 26 letters, we need to switch to *base 26*. This will ensure that each character's hash code will be contained within one digit place.

If the previous paragraph made perfect sense to you, you can skip the rest of this section. But if you're a little fuzzy on the details of number base systems, read on.

Take the number "ten." Note how I spelled out this number using alphabetical characters. If I wanted to express this same number using numerical digits, I'd write "10." However, this isn't the *only* way to express the number "ten" using numerical digits. Before I move on to the alternatives, though, let me expound a bit on the "normal" system.

Base 10: The "Normal" System

The way we write numbers day to day is known as the *decimal system*, and is also called *base 10*. The idea behind base 10 is that we have 10 different numerical *characters* available to us, namely, the character set `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`. No matter how large a number we want to express, we're only able to use characters from this set.

So, to express the numbers zero through nine, I can do so using the 10 different numerical characters available to me. But how do I express the number "ten" itself? If I wanted to do this using one digit, I'd need an eleventh type of character. I could use fancy characters such as § or £, or alphabet letters, or even emojis if I so desired, but I'd need some sort of additional character beyond the digits `0` through `9`.

But those characters don't exist in the base 10 system. So we're forced to express the number "ten" by using *multiple digits*, putting various digits in different *places*.

When we express the number "ten" as `10`, we're saying that there's `1` ten, and `0` ones. The number `25` means that there are `2` tens and `5` ones.

The same goes for representing the number one hundred. We can express the number `99` using two digits, but when we want to express a larger number, we've simply run out of ways in which we can use only two digits. So, we move on to the hundreds place, and write out `100`.

That's how the base 10 system works. However, there can be a base system using *any* number, such as base 8, or base 26, or base 457. While we don't generally use these other bases in day-to-day life, some alternative bases are used in specific applications, especially in the world of computers.

Base 2: The Binary System

One of the most well-known bases outside of base 10 is *base 2*. Indeed, base 2, also known as the *binary system*, or just *binary*, is the number system that computers understand best. Just as base 10 has *ten* different numerical characters, base 2 has *two* numerical characters. These characters are `0` and `1`.

With this system, like most other systems, "zero" is written out as `0`, and "one" is written out as `1`. However, expressing the number "two" presents a hurdle, as we only have the characters `0` and `1` available to us. And so, we need to start using *two* digits. However, whereas with base 10, the second place from the right is the tens place, in base 2, the second place is the *twos* place. So here's how we write out "two" in binary:

```
 twos   ones
place  place
  1      0
```

This means that there's one "two" and zero "ones." In other words, this is the number two.

To express the number three in binary, we'd write: `11`. That is, there is one "two" and one "one." When you have a two and you have a one, that makes three.

To express the number four, though, we have to introduce a *third* digit place, since the greatest number we can express using two digits is the number three. Now, this third place expresses how many fours there are. In other words, it's the *fours* place. So, "four" is expressed as `100`. Weird! In base 2, then, `100` is not "one hundred," but "four."

Here are a few more binary examples:

```
                sixteens   eights   fours   twos   ones
                 place      place    place   place  place
3     ⟶                                      1      0
5     ⟶                           1          0      1
15    ⟶                    1      1          1      1
21    ⟶          1         0      1          0      1
```

As mentioned, computers run primarily on base 2 since most computers store data in binary format. Whether it's words, images, videos, or songs, under the hood, they're all stored as binary numbers.

True story: I once bumped into an old friend, and when I told him that I'd become a software engineer, he asked me in all honesty whether I write code in zeroes and ones. I explained to him that—yes, of course—in fact, the keyboard I use has only two keys! He was quite impressed.

Base 16: The Hexadecimal System

Let's move on to another system, namely, *base 16*, which is also known as the *hexadecimal system*. The funny thing about the hexadecimal system is that 16 is greater than 10. This means that we have *sixteen* different numerical characters available to us.

Now, that may seem daunting, given that in real life we don't have numerical digits beyond `9`. But here's how we pull this off.

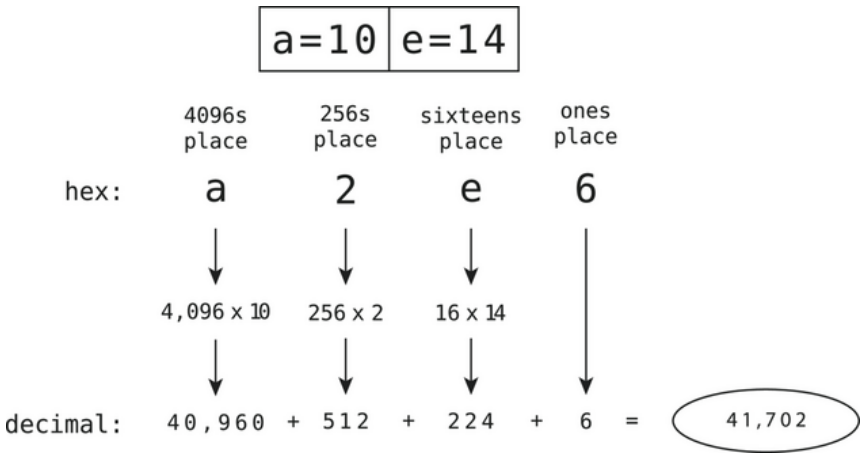
In hexadecimal notation, we have the following "numerical" characters available to us: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`. That is, `a` represents "ten," and `b` represents "eleven," and so on. The final character, `f`, represents "fifteen." That's right; in base 16, the letters `a` through `f` are *numbers*.

Of course, even base 16 has its limitations. When we want to express the number "sixteen," we've run out of single characters to use. And so, we move on to the next place over, which is the *sixteens* place. In hexadecimal, `10` is "sixteen." That is, there's one sixteen, and zero ones.

The greatest number we can express with two digits is `ff`, which is fifteen in the sixteens place, and another fifteen in the ones place. This comes out to two hundred fifty-five, as `(15 * 16) + 15 = 255`.

To express two hundred fifty-*six*, though, we need to move over to the next place, which is the two-hundred fifty-sixes place. And so, in hexadecimal, `100` is what in decimal we call `256`.

It can be a little hard to wrap one's mind around this, but the hexadecimal number `a2e6` is forty-one thousand, seven-hundred two. This is because:



You may have encountered hexadecimal numbers before. Very often, computer colors are expressed as hexadecimal numbers, otherwise known as *hex codes*. For example, `#2ECC71` is a lovely shade of green.

As I said, you can make a base out of *any* number. However, some of the most commonly used ones beyond the decimal system, especially in computing, are base 2 (binary), base 8 (octal), and base 16 (hexadecimal).

Before concluding this section, I want to point out something that holds true for *all* base systems. No matter the base system, we use the following scheme to determine what number each digit place represents. In the following visual, the `b` is a variable that represents the base number. So for base 2, the `b` is 2, and for base 10, the `b` is 10:

	...	b^4	b^3	b^2	b^1	b^0
base 2 (b=2)	...	16s place	8s place	4s place	2s place	ones place
base 10 (b=10)	...	10,000s place	1,000s place	100s place	10s place	ones place
base 16 (b=16)	...	65,536s place	4,096s place	256s place	16s place	ones place

That is, the right-most digit place in all base systems is b^0. Now, any number raised to the power of zero is 1, so the right-most digit place is always the ones place. But as we move leftward, each place is raised to the

next power. The table shows you how this plays out for bases 2, 10, and 16, but this pattern holds true for every base system.

Okay! We've covered the basics of base systems. Now, let's see why this matters for the Rabin-Karp algorithm.

Perfecting Rabin-Karp with Base 26

Let's quickly review the problem we encountered earlier. The Rabin-Karp algorithm worked fine when we were dealing with an alphabet of 10 characters. This is because each character, when converted into an integer, would live by itself in its own digit place within a larger number. With such a scheme, it was impossible for any hash code to represent more than one string, which is a good thing.

But as soon as we introduce more than 10 characters into our alphabet, we now have characters that convert to two-digit numbers, such as `k`, `l`, and `m`, which convert to 10, 11, and 12, respectively. And two-digit numbers take up, well, two digit places. And when this happens, we can no longer know if a two-digit number represents a single character that corresponds to a two-digit number, or if it corresponds to two single characters that each correspond to a single-digit number.

Additionally, we saw how a single hash code (that is, `120`) can represent both `"lk"` and `"ma"`, even though they both have the same number of characters. What we need is some way to ensure that each character will only take up one digit place, even after we hash it. Fortunately, we can use what we learned about numerical base systems to help us.

The String Is a Number

Before we look at how base systems will make things better, let's first take another look at strings, this time from an entirely different perspective.

If we chose to, we could look at any alphabetical string *as if it were a number*. Here's what I mean.

Let's work with our familiar scheme of `a = 0`, and `b = 1`, and so on. What hash code does the string `"cab"` convert to?

That's right, the string `"cab"` is equivalent to the integer `201`.

If we wanted to, we could say that the `"cab"` *is* the number `201`. We happen to be using alphabetical "digits" rather than numerical digits to express this number. This may sound like mere semantics, but trust me, it'll help.

Based on this perspective, we'd also say that until this point, our hash function has been viewing a string as a base 10 number and converting its digits from alphabetical ones into their numerical equivalents.

The problem, though, was that when our strings contained characters that came after the letter `j`, our strings made for terrible base 10 numbers. This is because when we represent a base 10 number using numerical digits, each numerical digit takes up one digit place. But when we use a string, many characters take up *two* digit places.

A New Base System for Strings

The trick to solving our problem is this: instead of treating a string as a *base 10* number, we should treat it as a *base 26* number. More generally speaking, the base system of a string should correspond to the number of characters in the alphabet we're using.

Let me spell out what it means to treat a string as if it were a base 26 number.

In base 26, here's what each digit place represents:

456,976s place	17,576s place	676s place	26s place	ones place

Let's now take the string `"bmm"`. (I know, it sounds pretty random, but wait for it …) If we treated this as a base 26 number, what number would it be? If you have the self-control of a yoga master, try to work this out for yourself before reading on.

Here's how `"bmm"` breaks down as a base 26 number:

456,976s place	17,576s place	676s place	26s place	ones place
		b (1)	m (12)	m (12)

In this image, and those that will follow, I've placed the corresponding decimal value below the letter. So, in this example, there is one `676` (represented by the "b"), twelve `26`s (represented by an "m"), and twelve ones (also represented by an "m"). This is the base 26 way to express the number *one-thousand*, since:

```
(1 * 676) + (12 * 26) + 12 =
676 + 312 + 12 =
1000
```

Now, if we treat our strings as base 26 numbers rather than base 10 numbers, we automatically solve the digit bleeding problem we struggled with before. A *bleeding* issue occurs when we have a character that needs more than one digit place to hold it. But because each digit place of a base 26 number can hold 26 different characters (such as `a` through `z`), and we only have 26 characters in our string's alphabet, we're guaranteed that each character can fit neatly into a single digit place.

A Base-26 Hash Function

Let's incorporate these ideas into Rabin-Karp and see how everything plays out. We'll go back to the beginning, starting with hashing our needle. In the example at the start of the chapter, the needle was `"cafe"`. Let's peek back at how we hashed `"cafe"` when we treated the string as a base 10 number:

a=0	c=2	e=4	f=5

```
c ⟶ 2 | x 1,000 = 2,000
a ⟶ 0 | x   100 =     0
f ⟶ 5 | x    10 =    50
e ⟶ 4 | x     1 =     4
                  -------
                    2,054
```

We're going to stick with this basic scheme, except that now we'll treat our strings as base 26 numbers. That is, we're going to view `"cafe"` as follows:

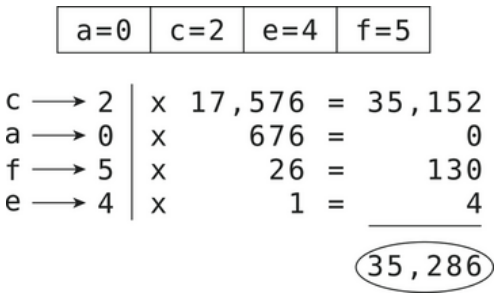
456,976s place	17,576s place	676s place	26s place	ones place
	c (2)	a (0)	f (5)	e (4)

This is the number that, in English, we'd refer to as thirty-five thousand, two-hundred, eighty-six.

Recall that our ultimate goal is to convert strings into Python integers so that our code can compare them more quickly. The thing, though, is that Python integers are represented as *base 10*. (It's binary under the hood, but when we write the code `x = 11`, Python interprets it as base 10, so `x` is eleven.)

This is what our hash function does: *we treat the string as a base 26 number, and convert it into a base 10 number*.

More specifically, instead of multiplying each digit by `1`, `10`, `100`, or `1000` like we did originally, we multiply each digit by `1`, `26`, `676`, and `17576`. And so, our hash function will hash `"cafe"` in the following way:



As you can see, when we hash `"cafe"` it yields a result of thirty-five thousand, two-hundred, eighty-six—or what we write in decimal as `35286`. Our problem is now solved. The string `"cafe"` can represent one and only one particular base 26 number. Our hash function simply converts that base 26 number to a base 10 number since that's what Python understands.

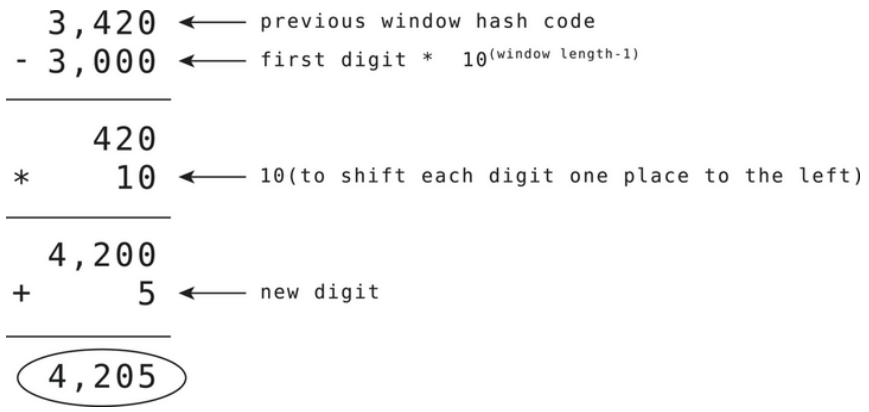
Conversely, we also know with certainty that the base 10 number `35286` can only represent the letters `"cafe"` and no other string. For when we convert a number from one base system to another, we're merely expressing the same number in different ways. And so, if we'd convert `35286` from base 10 to base 26, and use our scheme of `a = 0...z = 25`, we'd come up with the string `"cafe"`.

In our code, however, we're only going to be doing this conversion in one direction. That is, we never need to take a base 10 number and convert it back into base 26. The modus operandi of Rabin-Karp is that we convert both the needle and haystack window from base 26 to base 10 and see if they match.

Rolling with Base 26

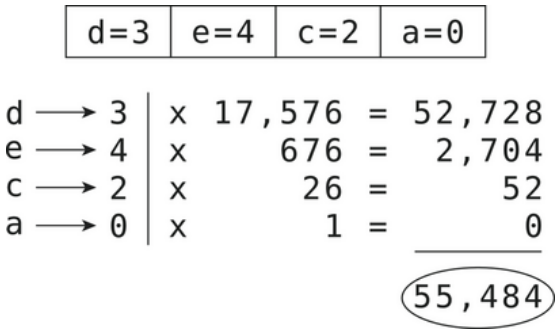
At this point, we've successfully adapted our initial hash function to handle alphabets of 26 characters. However, we've only addressed the *initial* hash function, that is, the one that hashes an entire string at once. We now have to modify our *rolling* hash function as well. Luckily, it's going to be a quick and easy modification.

As a reminder, here's what our rolling hash function did when working with base 10:

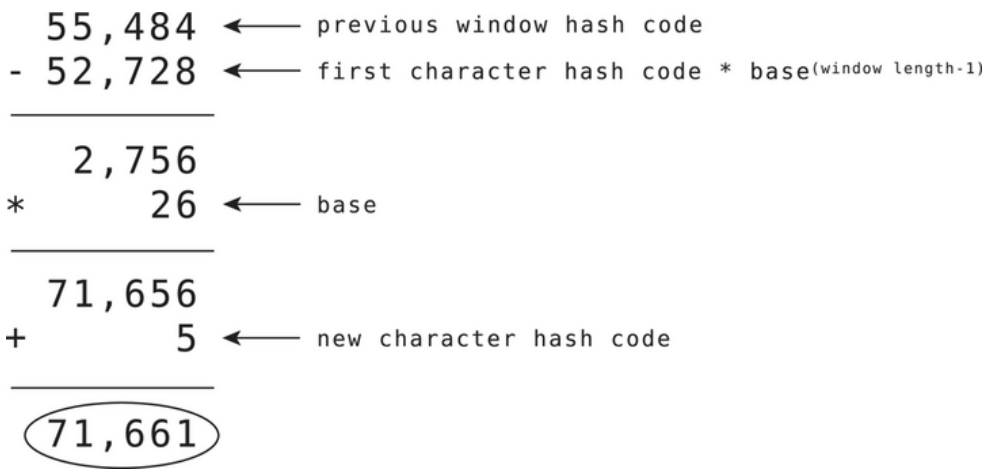


Again, this updates the hash code from the previous haystack window and converts it to match the current haystack window. With this math, we effectively remove the old first digit and tack a new one at the end. In this example, the `3420` lost the left-most `3` and gained a new `5` at the end, turning into `4205`.

To get this hash function to work with base 26, all we have to do is change the formula so that wherever we've been multiplying by `10`, we multiply by `26` instead. With our haystack of `"decafcafeahead"`, the first window is `"deca"`, which in base 26 hashes to `55484`, as you can see here:



Now, the next character in our haystack is an `"f"`, which corresponds with the digit `5`. Here's how our rolling hash formula will execute:



This gives us a new hash code of `71661`, which indeed is the base 26 representation of the new haystack window `"ecaf"`.

In other words, our old rolling hash function multiplied things by `10` because we were working with base 10 numbers. Since we are now working with base 26 numbers, we multiply by `26` instead. Let's walk through the example line by line to make it super clear.

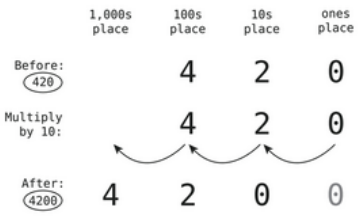
We start with the previous window's hash code. Our immediate goal is to delete the old `"d"` that our new window is dropping. Because the `"d"` in base 26 is in the 17,576s place, this computes to `52728`. So, by subtracting this from the previous hash code, we effectively delete that initial `"d"`.

Then, we shift all the digits to the left by multiplying our current result by `26`. When we worked with base 10, we multiplied by `10` to shift the digits. But now that we're working with base 26, we need to multiply by `26` to shift the digits. This might be intuitive to some, but I always like spelling things out (if you haven't noticed). In the next section, I'll elaborate further as to why this works.

Shifting Places

Back when we were dealing with base 10, by multiplying a number by 10, we were able to shift each of its digits one place to the left. For example, we

multiplied 420 by 10, which effectively shifted each digit one place leftward:



Essentially, when we multiply 420 by 10, we are saying to take the 4 hundreds and multiply them by 10, and to take the 2 tens and multiply them by 10. This gives us 4200.

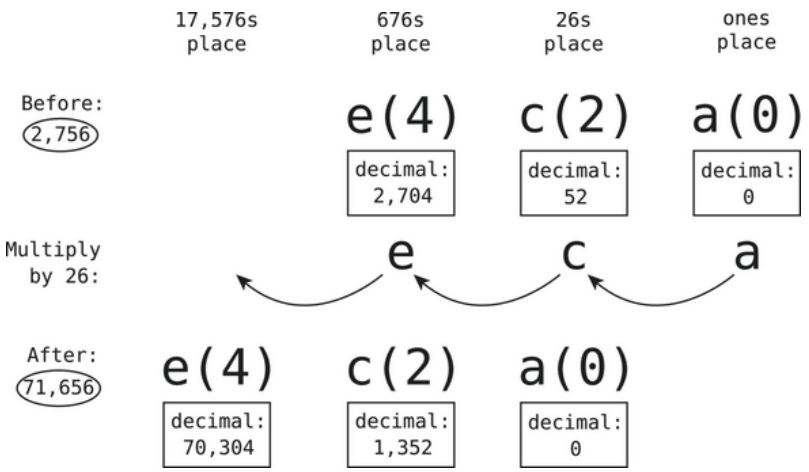
Now, here's the thing. To shift any number from *any* base in this way, we simply need to multiply the number by the *base itself*. So, because the previous 420 example was a base 10 number, we multiplied it by 10 to shift the numbers leftward.

To shift the digits of a base 2 number, we multiply the number by 2. And if our number is in base 16, we'd multiply the number by 16. This shifting process works no matter what the base is. The reason for this is outlined in the chart from earlier:

	...	b^4	b^3	b^2	b^1	b^0
base 2 (b=2)	...	16s place	8s place	4s place	2s place	ones place
base 10 (b=10)	...	10,000s place	1,000s place	100s place	10s place	ones place
base 16 (b=16)	...	65,536s place	4,096s place	256s place	16s place	ones place

Each digit place is equivalent to its right neighbor multiplied by the base. For example, the b^3 is the b^2 multiplied by the base ($b^2 * b = b^3$.) So when we multiply a number by its base, we multiply *each digit in each place* by the base, and thereby shift each digit to the left. And so, now that our hash functions work with base 26 numbers, we can shift such numbers by multiplying them by `26`.

Let's look at this in action with our example. After subtracting out the initial `"d"`, we had a resulting code of `2756`. Here's the breakdown of what happens when we multiply `2756` by `26`:



This visual shows how multiplying `2756` by the base of `26` effectively shifts each "digit" one place to the left.

Getting back to our rolling hash function, after we perform this shifting step, the final step is to tack on the current window's new character at the

end. We do this simply by adding the new character's hash code to the result of our previous computations.

We can now generalize our rolling hash function for any base, as follows:

```
  previous window hash code
- first character hashcode * base^(window length-1)
*                           base
+         new character hash code
-------------------------------------
      ( new window hash code )
```

Whew! I think we've made Mrs. Wilson proud.

Code Implementation: Rabin-Karp for Any Base

Let's modify our Rabin-Karp code so it handles base 26. Even better, let's make it handle *any* base:

```
# Global base variable:
base = 26


def find_needle(haystack, needle):
    needle_hash_code = initial_hash(needle)
    window_hash_code = initial_hash(haystack[0:(len(needle))])

    if needle_hash_code == window_hash_code:
        return 0

    for index in range(1, len(haystack) - len(needle) + 1):
        drop_character = haystack[index - 1]
        new_character = haystack[index - 1 + len(needle)]
        window_hash_code = rolling_hash(window_hash_code, len(needle),
                                        drop_character, new_character)

        if needle_hash_code == window_hash_code:
            return index

    return None
```

```
def initial_hash(string):
    power = 0
    result = 0

    for char in reversed(string):
        result += character_hash_code(char) * base**power
        power += 1

    return result


def rolling_hash(hash_code, window_length, drop_character, new_character):
    drop_number = \
        character_hash_code(drop_character) * base**(window_length - 1)
    result = hash_code - drop_number
    result *= base
    result += character_hash_code(new_character)

    return result


def character_hash_code(char):
    return ord(char) - 97
```

This code is almost identical to the previous implementation, with the only difference being that we've made the base flexible instead of hardcoding it as `10`. To do this, we declare a global `base` variable, which is then used within our hash functions.

Currently, `base` is set to `26` since we're working with an alphabet containing 26 characters. If you were working with, say, the entire range of ASCII characters, you'd set `base` to `256`.

Handling Long Needles

Armed with our new, shiny implementation of the Rabin-Karp algorithm, we're ready to tackle substring search for an alphabet of any size. This could include uppercase letters, lowercase letters, the numbers 0 through 9, punctuation marks, and various other characters. For our discussion, though, we'll continue to use the 26 lowercase letters of the English alphabet.

But there's a problem.

In our example, we had used a pretty short needle. After all, `"cafe"` only contains four characters. But it's not uncommon to search for longer needles. If we're searching a document for plagiarism, for example, we might use an entire sentence as a needle.

Now, say that our needle is the 28-character word `"antidisestablishmentarianism"`. If we hash this from base 26 to base 10 (using our scheme `a=0`, `b=1`, and so on), we get the number: `846022787623077614691775512079479705 04`. That's pretty long. As our needle length increases, our hash codes will grow longer and longer. Imagine if we had a needle with a length of *500*. Yikes!

The reason this is problematic is that the entire reason we've been hashing words into numbers was so that a computer can compare two numbers quickly. However, this is, in fact, only true when the numbers are relatively small. When numbers are extremely large, computation speeds go down, and memory consumption goes up. Python is better than some other programming languages at processing large numbers, but even Python has its limitations.

Now, if your application is not going to be dealing with long needles, our current Rabin-Karp implementation is perfect, and there's no need to adjust

it. The remainder of this chapter is all about optimizing Rabin-Karp to handle both long and short needles efficiently.

Division Hashing

Fortunately, we can solve this issue with the technique of division hashing described in *The Division Method*. Specifically, we'll update our hash functions to not only convert numbers from base 26 to base 10, but *also divide that result by some prime number*. Let me show you what I mean.

The Basic Division Approach

The basic approach is that we'll tweak our hash function ever so slightly. Currently, our hash function is simply to convert base 26 to base 10. But again, the problem is that if our string is very long, the hash code will be very long as well.

Our trick is to perform one more computation after converting from base 26 to base 10. That is, after we do the base-26-to-base-10 computation, we then divide the result by a predesignated prime number and grab the remainder. *This remainder is now our hash code*. Later, I'll discuss how we'll go about choosing our prime number. But whatever prime number we choose, this will be the one we use throughout our entire substring search algorithm.

Let's walk through an example.

Say that our prime number is `613`. When we divide a number by `613`, the remainder will be some number from `0` up through `612`.

Let's now hash our needle, `"cafe"`. Earlier, our hash code of converting base 26 to base 10 gave us a result of `35286`. We can now update our hash code so that we then take the `35286` and perform one more computation of:

```
35286 % 613 = 345
```

With this update, the hash code of `"cafe"` is now the shorter `345` rather than the longer `35286`. What's nice about this is that even if our needle was much longer, such as `"antidisestablishmentarianism"`, the hash code will never be larger than `612`. Indeed, the hash code of `"antidisestablishmentarianism"` is now:

```
846022787623077614691775512079479705O4 % 613 = 230
```

Computer scientists like to refer to this concept as *fingerprinting*. A detective can use a fingerprint to determine which person touched the doorknob, even though the fingerprint is much smaller than the actual person. Similarly, we can use a small piece of data (such as `230`) to represent a larger piece of data (such as `846022787623077614691775512079479705O4`).

So far, we've updated the initial hash function that computes the hash code of the needle and the first haystack window. But we're going to use this same general division approach for our rolling hash function as well. That is, we'll be dividing various results by the same prime number of `613`. Later, I'll explain how precisely to perform this math, but let's take this general approach at face value.

Because we're applying the same hash function—including the division—to our needle as well as the haystack windows, we can still perform an effective substring search. That is, if the needle's hash code is the same as the haystack window's hash code, we've found our match. Otherwise, the two strings are definitely not the same.

Collision Ahead

This is where things get interesting and is, in fact, the impetus for discussing the Rabin-Karp algorithm at this point in the book.

Before we introduced division hashing into our Rabin-Karp algorithm, the hash function guaranteed that two strings must be identical if they have the same hash code. But now that we started shortening numbers with division

hashing, it's possible for two *different* strings to end up with the *same* hash code. As we've seen, *all* numbers, no matter how large they are, will end up having a hash code in the range of `0` through `612`. If you take, say, 614 different strings and hash them, it's guaranteed that at least two of them will end up with the same hash code. Likely, many more strings will share hash codes as well.

We saw this same idea when developing hash functions for hash tables. There's always a possibility that more than one value will end up within the same hash table slot.

Now, for hash tables, this may not be the biggest deal since we can handle collisions with separate chaining and other techniques. But it *is* a real problem for substring search. That is, our code might mistakenly think that it found the needle in the haystack even when it didn't!

Here's a quick example. We already know that our needle `"cafe"` hashes to `345`. Now, if our algorithm encounters the haystack window `"clzr"`, the computer will hash it by converting it from base 26 to base 10, which yields `43255`. Then, the computer will divide it by `613` and grab the remainder, which is ... `345`! The computer will then report that it successfully found `"cafe"` when this couldn't be farther from the truth.

It's worth pointing out that the opposite type of bug *cannot* occur. That is, the computer will never encounter a match and erroneously report that it's *not* a match. If two strings are, in fact, identical, they most certainly will have the same hash code. But we do have the potential error in the direction of the computer thinking that it's found a match when it actually hasn't.

However, let's analyze the likelihood of this error happening.

Monte-Carlo Rabin-Karp

When we hash a value by dividing it by a number, the resulting hash code will be some number from `0` up to (but excluding) that number itself. So if our chosen prime number is `613`, the hash code will be some number from `0` through `612`. Put another way, there are 613 possibilities as to what the final hash code will be.

It turns out that when a haystack window does *not* match the needle, there's still a 1 out of 613 chance that their hash codes will match. If we choose a larger prime number, such as `7841`, then each haystack window has a 1 in 7,841 chance of being incorrectly identified as a match to the needle.

Now, if our haystack is small relative to our needle so that we only have a few haystack windows, we're unlikely to encounter any mistaken matches. For example, if each haystack window has a 1 in 7,841 chance of being a mistaken match, if we have 3 haystack windows to check, there will be a 3 in 7,841 chance that we'll encounter a mistaken match over the course of our entire search.

As our haystack grows, though, the odds of a mistake increase. If we have, say, *8,000* haystack windows, then there's a decent chance that we *will* come across an incorrectly identified match at some point in our search.

The trick for keeping the probability of a mistake low is to select a high prime number. If we could pull off the math to select an ideal prime number like this, it becomes significantly improbable that a mistake will occur.

And this, my friends, is what makes Rabin-Karp a Monte Carlo algorithm. Recall that the whole point of Rabin-Karp was to make substring search much faster than the brute-force approach. To accomplish this, though, we end up accepting a small possibility that the algorithm will not produce an accurate result. This is precisely what Monte Carlo algorithms do: they

sacrifice accuracy for the sake of increasing speed. If we ensure that our prime number is high enough, we'll end up with a speedy substring search with a low chance of error.

It emerges that if our app isn't going to handle long needles, we can avoid collisions altogether if we use base 26 and do *not* use division hashing. It's only if we need to be concerned with long needles that end up in this situation where two different strings may end up with the same hash code.

As things stand now, the current variant of Rabin-Karp is a clever Monte Carlo algorithm. But there's a plot twist.

Converting Monte Carlo to Las Vegas

It turns out that it's possible to *convert certain Monte Carlo algorithms into Las Vegas algorithms*. Not all Monte Carlo algorithms can be transformed this way, but Rabin-Karp is an algorithm that can. As a quick reminder, while a Monte Carlo algorithm is an algorithm that is definitely fast with a small probability of being incorrect, a Las Vegas algorithm is definitely correct, with a small chance of being slow.

Currently, Rabin-Karp will definitely be fast, but may be incorrect. We will now tweak the Rabin-Karp algorithm so that it becomes definitely correct, but will have a small chance of being slow.

This tweak is simple. We'll add one additional step to the algorithm. Specifically, each time our algorithm finds matching hash codes, the algorithm will then do a "sanity check"; it will check both strings the old-fashioned way, character by character, to see if they're the same.

If our needle is `"cafe"`, and the haystack window is `"clzr"`, which both result in the hash code `345`, the algorithm will next do a sanity check by comparing all the characters of these two strings. In this case, the strings are *not* the same, so the algorithm will know that it hasn't yet found a true match and will move on and continue to search the rest of the haystack for our needle. By performing this extra check, the computer will *never* report a false match. Rabin-Karp is now a Las Vegas algorithm since it will definitely be correct. And as we'll now see, the odds of it being slow are also slim.

Each time the computer performs a sanity check to see if two strings are the same, this is relatively slow, as it has to comb through each character of both the haystack window and the needle. And if the algorithm ends up finding numerous instances of matching hash codes, this will require numerous sanity checks and will slow down the entire algorithm.

But here's what's interesting. What are the *odds* that the computer will slow down?

It turns out that the odds that the computer will perform a sanity check (when the two strings are indeed not the same) are the *very same* odds as the Monte Carlo version of Rabin-Karp not being correct.

That is, we noted that if the prime number is `7841`, then each haystack window has a 1 in 7,841 chance of producing a false match. Well, in the Las Vegas form of the Rabin-Karp algorithm, there's that same 1 in 7,841 chance that the algorithm will encounter a false match at this point and perform that "slow" sanity check. Even if the algorithm has to perform a handful of sanity checks here and there, we likely wouldn't notice a slowdown. The algorithm will only slow down significantly if it's finding *many* false matches and performing sanity checks over and over again. But again, this is unlikely if each haystack window has only a 1 in 7,841 chance of being a false match.

This new version of the Rabin-Karp algorithm is truly a Las Vegas algorithm. It is definitely correct and has only a small chance of being slow.

The moral of this story is that when you do encounter a Monte Carlo algorithm, see if there's a way to transform it into a Las Vegas algorithm. This transformation isn't always possible, but if it is, it might be worth considering.

At this point, we're almost through with everything we need to know about Rabin-Karp. The final item is to work out the math of the hash function so that it uses division hashing. I happen to think that the math that follows is pretty cool, but if you've had enough math for the day, I understand completely. You can skip ahead to the end of the chapter if you prefer.

Modulus Magic

Here's a quick summary of where we're at right now.

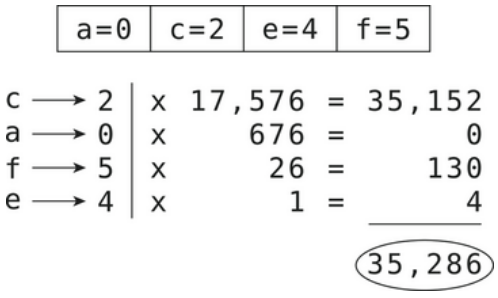
Previously, our Rabin-Karp hash function simply treated each string as a base 26 number and converted the number into base 10. In truth, this worked pretty well. It was guaranteed to be both accurate *and* fast and certainly got the job done. The only problem was that it would not perform well in cases where the needle length is very long; the hash function would produce large numbers, which would slow down the entire algorithm. If not for the possibility of long needle lengths, I could have ended the chapter at that point.

But if we want to make the Rabin-Karp algorithm production-ready, we want it to perform well for long needle lengths. That's why we introduced division hashing, which takes large numbers and cuts them down to size. Specifically, it ensures that the hash code will never be larger than the prime number we use in our division.

However, to pull this off, we'll need to employ some clever mathematical tricks, and here's why.

With division hashing, we first compute a long hash code, and then divide it by a prime number (and grab the remainder). But if the needle is super long, *it will also take a long time to compute that long hash code.* In other words, we'll encounter a slowdown before even getting a chance to perform our division!

To spell this out even further, recall that before division hashing, we hashed the needle this way:



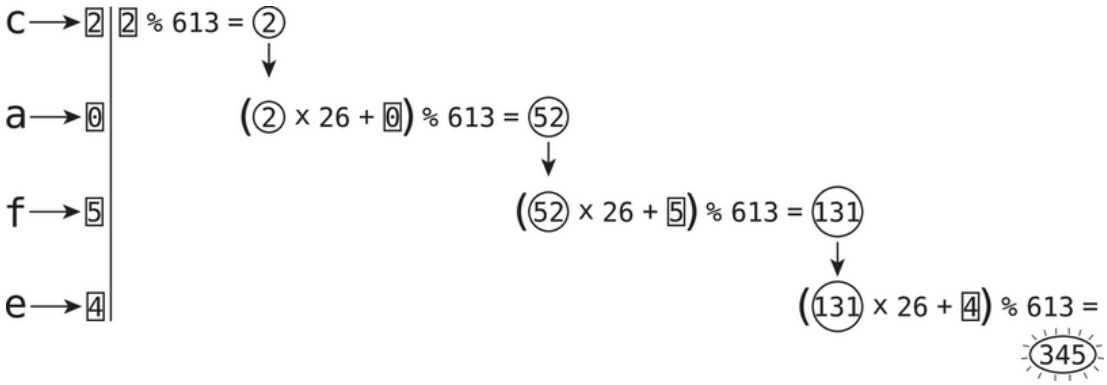
In this case, we multiply the `"e"` by 1, the `"f"` by b^2 (with "b" being 26), the `"a"` by b^3, and the `"c"` by b^4.

Imagine, now, that a needle was 500 characters long. This means we'd have to continue to compute each subsequent character by b^5 and b^6 all the way up to b^{500}! Those end up being some pretty large numbers; for example, 10^{500} is 1 followed by 500 zeroes.

And therein lies the problem. Although computers are fast, they have their limitations, and computing large numbers can be significantly slow. To resolve this issue, we're going to use some ideas from the field of *modular arithmetic* (math having to do with modulus operations) that some very clever people figured out and applied to our situation.

Let's keep going with our `"cafe"` needle example. As we've seen, the initial hash code, before the prime number division, is `35286`. To keep our examples simple, let's continue to use a smallish prime number, namely, `613`. When we compute `35286 % 613`, we get `345`.

But watch this amazing alternative way to compute the same result. It's based on the mathematical properties of modulus operations. Here goes:



Here's what's going on in this computation. Instead of doing one single modular operation at the end of the hash function, we instead perform a modular operation for *each and every character*.

In this example, we start with the `"c"`, grab its hash code (`2`), divide by the prime number, and grab the remainder. We then take that remainder (`2`), multiply it by the base (`26`), add the new character's hash code (the hash code of `"a"` is `0`), and perform the modulus operation again. We repeat this process for each character of our string.

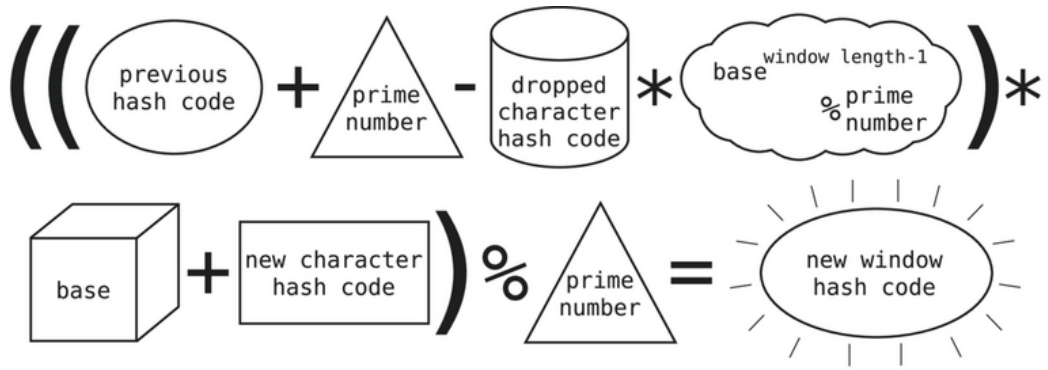
Throughout this process, we never have to deal with any number larger than `613`, and we still end up getting our desired result. Cool!

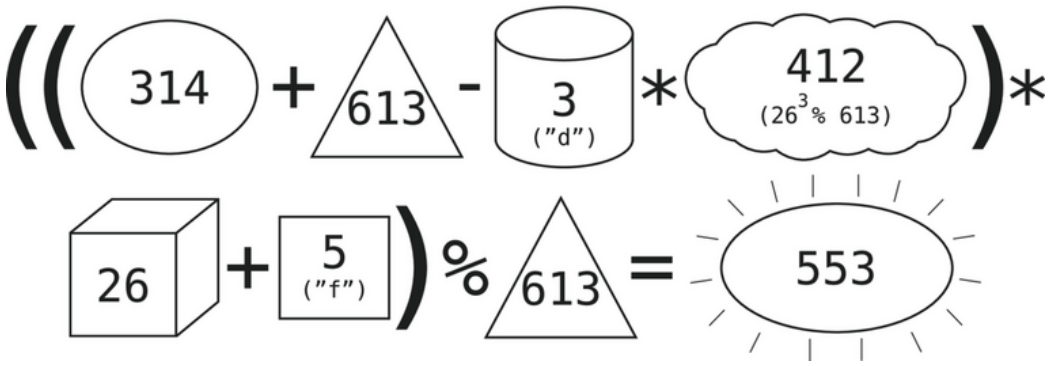
At this time, I'm not getting into *why* this modular arithmetic trick works. For now, let's just marvel at the magic of math and take this at face value.

We can use the earlier trick as is to serve as our *initial* hash function. However, we do need to update our *rolling* hash function so that it incorporates this type of math magic as well.

Let's go back to our example haystack. If we perform our modified initial hash function on the first haystack window of `"deca"`, we get the hash code `314`. We now need to perform our rolling hash function on the next window, which drops the initial `"d"` and introduces the character `"f"`.

Here's the math formula for the rolling hash function. I'll admit, it's a little involved. But it works! On the top, I show the actual formula, and on the bottom, I show how it applies to our example by plugging in all the numbers:



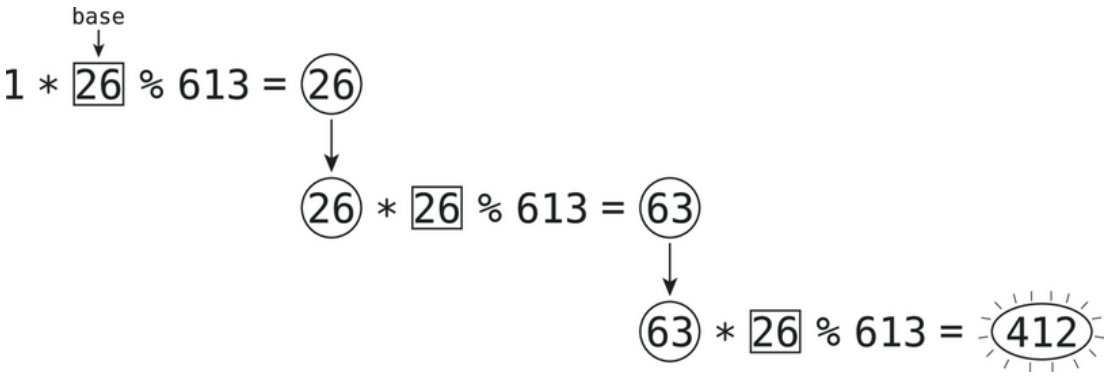


Here, we successfully compute the hash code for the new window `"ecaf"` based on the previous window of `"deca"`. The result of `553` is the same result we'd get if we first converted `"ecaf"` to `71661` (by converting base 26 to base 10) and then dividing `71661` by our chosen prime number of `613`. That is, `71661 % 613 = 553`. However, with this fancy formula, we get the identical result without ever needing to first come up with the longish number of `71661`.

Now, there's one last bit of math that we need to perform. In the visual, take a look at the item shaped as a cloud. As you can see, this is a number computed by taking $base^{(window\_length-1)}$, dividing by our prime number, and grabbing the remainder. I call the result of this computation the "drop place remainder."

However, we once again run into a problem. If our window size is `500`, this means that when we first compute $base^{(window\_length-1)}$, we end up with a large number and a slow computation.

However, we can use modular math tricks to compute this number without ever having to wrangle with large numbers. Here's how:



Here's what the formula does:

We start with the number 1 and multiply it by the base (such as 26). We take the result, divide it by our chosen prime number, and grab the remainder. We then take this result and repeat the same process as many times as the length of the needle minus 1. (In this visual, we're working with an example of where the needle length is 4.)

This computation produces the drop place remainder without ever producing a number larger than the prime number.

Whew! We've come to the conclusion of how Rabin-Karp works in all its gory detail. Let's go ahead and code it all up.

Code Implementation: Las Vegas Rabin-Karp Substring Search with Division Hashing

Here's the Python code for the Rabin-Karp algorithm with division hashing incorporated. Additionally, we've also transformed it into a Las Vegas algorithm:

```
# Global base variables:
base = 26
prime = 613
```

```
def find_needle(haystack, needle):
    needle_hash_code = initial_hash(needle)
    window_hash_code = initial_hash(haystack[0:(len(needle))])

    if needle_hash_code == window_hash_code:
        return 0

    # Precompute the "drop place remainder":
    drop_place_remainder = 1
    for i in range(len(needle) - 1):
        drop_place_remainder = (drop_place_remainder * base) % prime

    for i in range(1, len(haystack) - len(needle) + 1):
        drop_character = haystack[i - 1]
        new_character = haystack[i - 1 + len(needle)]
        window_hash_code = rolling_hash(window_hash_code,
                                        drop_character, new_character,
                                        drop_place_remainder)

        if needle_hash_code == window_hash_code:
            # Las Vegas sanity check:
            if needle == haystack[i:(i + len(needle))]:
                return i

    return None


def initial_hash(string):
    result = character_hash_code(string[0]) % prime

    for i in range(1, len(string)):
        result = (result * base + character_hash_code(string[i])) % prime

    return result


def rolling_hash(hash_code, drop_character, new_character,
                 drop_place_remainder):
    result = ((hash_code + prime - character_hash_code(drop_character) *
               drop_place_remainder) *
               base + character_hash_code(new_character)) % prime

    return result
```

```
def character_hash_code(char):
    return ord(char) - 97
```

The thrust of this code is similar to previous implementations. The difference, though, is that the hash functions themselves have now changed since they incorporate division hashing according to the formulas I described earlier.

Because the new hash formulas rely on a predetermined prime number, we set a global `prime` number variable at the top. I've set it to `613` to correspond to our previous examples, but in real life, this would likely be a larger number. In fact, you can also choose it randomly based on the Monte Carlo approach described in *Monte Carlo Primality Testing*.

As part of the new division hashing approach, we precompute the `drop_place_remainder` within the `find_needle` function. Once we've computed the `drop_place_remainder`, we use it as part of the `rolling_hash` function. (The `initial_hash` function doesn't need it.) As to the hash functions themselves, they're more involved now, but track directly to the earlier formula descriptions.

The one final novelty of this implementation is that we transform the `find_needle` code into a Las Vegas algorithm with a single extra line of code. That is, if the `needle` and `haystack` window hash codes match, we do a sanity check and see if the two strings themselves are indeed identical:

```
if needle_hash_code == window_hash_code:
     if needle == haystack[i:(i + len(needle))]:
          return i
```

The code `haystack[i:(i + len(needle))]` represents the haystack window. This sanity check takes the computer up to as many steps as there are characters in the strings. But again, the odds are low that we'll have to perform many sanity checks.

Well, we've done it!

Division Hashing for Strings

In the previous chapter, I explained how division hashing works. However, I focused solely on hashing integers and held off on the discussion of applying division hashing to strings. (I did bring it up in the exercises, though—and of course, you knew that.)

It turns out that one of the most efficient ways to use division hashing for strings is the `initial_hash` method from our final implementation, which went like this:

```
def initial_hash(string):
    result = character_hash_code(string[0]) % prime

    for i in range(1, len(string)):
        result = (result * base + character_hash_code(string[i])) % prime

    return result
```

Besides being fast, it also ensures that anagrams don't necessarily produce the same hash codes as each other, something which is not the case if we simply convert each character to a number and multiply the numbers together.

So, in the `DivisionHasher` class we implemented in *Code Implementation: Randomized Hash Functions*, we can swap out the `hash` method's code and instead use the code from the `initial_hash` method. That's the gist of it, anyway. I'll leave the exact implementation to you, as an exercise to follow.

Wrapping Up

The Rabin-Karp algorithm uses a number of the techniques we've learned about in the past few chapters. From division hashing to Monte Carlo, we've pulled out all the stops. And the result is phenomenal: we've achieved linear-time substring search.

We also looked at the important concept of base number systems, which will also be crucial for the upcoming chapters. Finally, we've also seen how sometimes it's possible to convert a Monte Carlo algorithm into a Las Vegas algorithm and eliminate the chance for any inaccuracies.

I recommend that you also check out other linear-time substring search algorithms such as KMP and Boyer-Moore, as they're fascinating and use a completely different approach than Rabin-Karp. Indeed, each of the three algorithms has nuanced pros and cons. One of the pros of Rabin-Karp is that it doesn't consume any extra space, while the other two algorithms do.

Over the past few chapters, you've learned how Monte Carlo algorithms can be used to speed up your code. I'll have you know, though, that Monte Carlo algorithms can also be used to *save space*, as I'll discuss in an upcoming chapter about Bloom filters. However, to understand Bloom filters, there's a prerequisite concept I need to cover first—and that is the data structure called the *bit vector*.

Exercises

The following exercises provide you with the opportunity to practice with string matching and the sliding window technique. The solutions to these exercises are found in the section *Chapter 11*.

1. Implement a new version of the `DivisionHasher` from Chapter 10 so that it can now hash strings instead of integers. Use the approach of the `initial_hash` function from our final implementation of Rabin-Karp. (I warned you that this would be an exercise!)

2. Here's a sliding window exercise for you. Write a function that accepts a string. The function should return the maximum number of vowels that are contained within any three-character substring (of the original string).

   For example, the correct answer for the string `"spider"` is `2` since the three-character substring `"ide"` contains 2 vowels. There's no three-character substring within `"spider"` that contains a greater number of vowels than 2.

   On the other hand, the correct answer for the string `"beautiful"` is `3` since the substring `"eau"` contains 3 vowels.

   For the purposes of this exercise, we'll say that vowels are the letters `a`, `e`, `i`, `o`, and `u`.

3. *New Concept:* In this chapter, when discussing the sliding window technique, we always worked with a window of a fixed size. That is, as the window slides, it always contains the same number of values. However, there's another form of the sliding window technique in which the window doesn't just slide, but also expands or contracts. Here's such a problem:

Write a function that accepts a string, and returns the *length* of the longest *substring* that doesn't contain any duplicate characters.

For example, in the string `"ababcdabca"`, the longest substring that doesn't contain any duplicate characters is `"abcd"`, starting at index 2. It has 4 characters, so our function should return `4`.

(There's another substring with the length of 4, namely, `"bcda"`, which starts at index 3. In any case, though, there's no substring that has a length *greater* than 4.)

Chapter 12

Saving Space: Every Bit Helps

Over the past several chapters, you've discovered a number of Monte Carlo algorithms. In the chapter that follows this one, you're going to look at your first Monte Carlo *data structure* called the Bloom filter. The Bloom filter, though, is based on another data structure called a *bit vector*, which in turn is based on yet another data structure called a *Boolean array*. In this chapter, I'll introduce you to both the Boolean array and the bit vector.

Both Boolean arrays and bit vectors, though similar, offer different types of performance boosts. In particular, Boolean arrays optimize for *time*, while bit vectors optimize for *space*. In either case, the respective benefits of each of these data structures are pretty astounding.

Along the way, we'll take a look at *bit manipulation*, which refers to a set of techniques that enable us to access the individual bits of an integer. Bit manipulation is a foundational computer science concept that has many useful and amazing capabilities, as you'll soon see.

Sets

In Volume 1, Chapter 1, I introduced the notion of a *set*. A set is a collection of unique values; there are no duplicate values. There are many different applications where maintaining a set can be useful, which I demonstrate in the following sections.

Finding Duplicates

Suppose we have an array of integers in which there may or may not be duplicate values, and we need a function that will tell us whether there are any duplicates. For example, if we have the array `[4, 3, 5, 1, 7, 2, 6, 8]`, our function will return `False` because no duplicate values are present. However, if the array is `[1, 2, 3, 1]`, the function will return `True` because there are two instances of the integer `1`.

The brute-force approach to detecting duplicates would be to use nested loops. The outer loop would iterate over each value, and for each value, we'd initiate a second loop that would scan the rest of the array to see if that value appears again within the array. Naturally, this approach has a speed of $O(N^2)$.

However, if we use a *set-building* algorithm (and the right data structure to serve as our set), we can get the job done in O(N) time. That is, throughout our algorithm, we'll maintain a set of all the values we've ever encountered before. With this approach, we can scan all N integers once, following these steps:

1. We check the current integer to see if it's a key in the set.

2. If it is, that means we've encountered this value before, which means the current value is a duplicate. So, our function returns `True`.

3. If the current value is *not* in the set, we insert it into the set. The set is a hash table that stores each integer as a key and `True` as the value. So, if our data is `[4, 3, 5]`, our hash table will be `{4: True, 3: True, 5: True}`.

4. If we search the entire array without finding a duplicate, our function returns `False`.

Assuming that each value in our set can be accessed in constant time, this algorithm has a time complexity of O(N). That is, in the "worst case," which is when there are no duplicates, we scan each of the N elements once. Although we also perform additional steps such as inspecting the set for the element and inserting the element into the set, this is 3N steps, which reduces to O(N).

Because the speed of this algorithm hinges upon having a set where each element can be accessed in constant time, we need to make a careful decision as to what data structure we'll use to house our set.

Choosing the Right Set Data Structure

In Volume 1, Chapter 1, I talked about using an array to serve as a set. An array would not be ideal for our scenario of duplicate checking, since we'd have to perform a linear search on the array each time we look up a value. This means that accessing our set has a cost of O(N) time, and we're seeking to achieve O(1) lookup time.

We'd do much better with a hash table, which offers O(1) lookups. That is, we'll store our integers in the hash table as keys and use any arbitrary truthy value, such as `True`, as the hash table values. Indeed, hash tables are often used to represent sets, especially where the situation warrants many lookups.

With this in mind, here's the code for detecting duplicate values where we use a hash table set:

```
def has_duplicates(array):
    set = {}

    for item in array:
        if set.get(item):
            return True
        else:
            set[item] = True

    return False
```

This code takes O(N) time, as we iterate over each of the N `array` items once. This may not be earth-shattering; we wrote this same code back in Volume 1, Chapter 19. However, I'm leading up to an important point, so bear with me.

In any case, this was one simple application where maintaining a set was useful. Let's look at one more.

Counting Sort

As I've mentioned numerous times, the fastest sorting algorithms take O(N log N) time for average-case scenarios. However, this isn't *quite* true for every application. We can sort certain data sets, believe it or not, in considerably faster time.

Suppose we have an array of integers and we have some special knowledge about the nature of these integers. For example, we may know that all of these integers fall within the range of `0` to `9999` and that there are no duplicates.

Armed with this knowledge about our data, we can use a set to help us sort the integers in *linear time*. Here's how:

1. We create a new empty array that will eventually contain all the sorted integers. (Alternatively, we could have chosen to overwrite the original array; it's easier to explain the algorithm this way.)

2. We scan all the integers and build a set—using a hash table—that stores each integer as a key and `True` as the value. So, if our data is `[860, 2345, 9999]`, our hash table will be `{860: True, 2345: True, 9999: True}`.

3. We then start a loop that I'll call the "counting phase." The loop will run 10,000 times, keeping track of a variable called `number`, which, before the loop begins, starts out as `0`. In each round of the loop, we increment `number` by 1. In the loop's final iteration, `number` will be `9999`.

4. In each iteration, we look up `number` inside our hash table. If it is there, this means that `number` is also in the original array. As such, we append `number` to the end of our *new* array.

And that's it! In other words, we only need to execute two simple loops. The first loop takes all of our integers and inserts them into a hash table. The second loop then counts from `0` to `9999`—that is, the variable `number`—and checks to see if `number` is inside the hash table. (If it is, we append `number` to our result array.) The reason this effectively sorts our array is that because we're counting `number` from `0` to `9999` in *sorted order*, our final result array will also be in sorted order.

This algorithm, which many computer scientists call *counting sort*, doesn't have to compare any of the values we're sorting to each other, which is what slows all the other sorting algorithms down. Here's the code for counting sort:

```
def counting_sort(array):
    sorted_array = []
    set = {}

    for value in array:
        set[value] = True

    for number in range(10000):
        if set.get(number):
```

```
                sorted_array.append(number)

        return sorted_array
```

As you can see, we use a hash table to serve as our set. Again, this allows us to look up each integer in constant time.

The Time Complexity of Counting Sort

The counting sort algorithm first inserts all N integers of the array into the hash table. It then performs a fixed number of 10,000 steps, looking up the numbers `1` through `10000` in the hash table. If, say, we had 5,000 integers that lie in the range of `1` through `10000`, the algorithm would perform a total of 15,000 steps. That is, it inserts the 5,000 integers and then performs 10,000 hash table lookups.

To generalize counting sort's time complexity in terms of Big O, we can use the variable N to refer to the number of integers in the array, and the variable R to refer to the size of the range of data. (In our current example, N is 5,000, and R is 10,000.) Armed with these variables, we'd say that counting sort has a speed of O(N+R).

Had we gone with Quicksort or Mergesort, though, sorting would have taken, on average, 65,000 steps. That is, these algorithms take O(N log N) time. In this example, N is 5,000, and log N is 13, so `5000 * 13 = 65000`.

This serves as another example of how sets, assuming they have O(1) lookups, can significantly speed up our code.

Before moving on, it is important to remember that counting sort isn't always the best choice. If, for example, our integers could lie in the range of 1 to 1,000,000, counting sort would end up taking at least 1,005,000 steps. Because we only have 5,000 values in our array, Quicksort would still have taken 65,000 steps. So, make sure that counting sort is a good fit for your data set before blindly using it. The shorter the range, the better fit that counting sort may be.

It's also worth noting that counting sort can also work even if our array contains duplicate values. That is, instead of setting each hash table value simply to `True`, we'd instead set it to be an integer representing the tally of how many times we encounter that value. That is, the first time we encounter a particular integer, we set its hash table value to `1`. When we encounter it a second time, we increment the corresponding hash table value to `2`, and so on. So, if the data was `[7, 1, 1, 1, 4, 4]`, our hash table would end up being:

```
{7: 1, 1: 3, 4: 2}
```

When we get to the number `4` during the counting phase, we'll add *two* `4`'s to our result array because the hash table tells us that there are two of them.

Later, I'll make an important distinction between counting sort, where the data contains duplicates, vs. where it does not. For now, though, keep in mind that counting sort can work for both scenarios.

Boolean Arrays

We've looked at a couple of examples where sets offer considerable speed benefits. In particular, we took full advantage of a hash table's O(1) lookups to achieve some really fast code.

The interesting thing, though, is that there's an even faster set data structure than a hash table. Don't get me wrong; hash tables are great for serving as sets. But in certain scenarios, other data structures can be even faster. One such data structure is the *Boolean array*. The name may sound fancy, but it's simply a regular array in which we only store the values `True` or `False`. The array `[True, True, False, True, False, False, True]` is a Boolean array.

A Boolean array can serve as a set when our data consists of integers that lie in a relatively limited range—similar to the earlier counting sort example. With such a data set, we can use the array's *index* to signify one of the integers of our data. For example, suppose we want to store a set that contains the integers `1`, `4`, `5`, and `9`. We can do so with a Boolean array that looks like this:

```
[False, True, False, False, True, True, False, False, False, True]
```

If you look carefully, you'll see that the indexes `1`, `4`, `5`, and `9` are all set to `True`. This is a simple trick for using a Boolean array to represent a set of integers.

Using a Boolean Array to Find Duplicates

Let's now go back to the previous examples of finding duplicates and counting sort. We'll swap out the hash table for a Boolean array and see what happens. We'll start with finding duplicates. The code is almost identical to our previous implementation. It's just that now we use an array instead of a hash table:

```
def has_duplicates(array):
    set = [False] * 1000

    for item in array:
        if set[item]:
            return True
        else:
            set[item] = True

    return False
```

We initialize our array by making it a Boolean array containing 1,000 `False` values. We've made it `1000` with the assumption that our data will consist of integers that lie in the range from `0` to `999`. If your data has a different range, you'd update this number accordingly.

One might reasonably assume that the efficiency of this code should be the same as when we used a hash table. With either a hash table or a Boolean array, in a worst-case scenario, we iterate over all N values of the input `array` and insert them into a set. Yet, when I benchmark the two competing code snippets, I get different results.

This is the speed of our hash table version over the course of five runs:

```
[0.0001630783081054687 5, 0.0001590251922607422, 0.0001559257507324218 8,
0.0001559257507324218 8, 0.0001559257507324218 8]
```

That's pretty fast! But look at my results of benchmarking the Boolean array code:

```
[7.295608520507812e-05, 7.605552673339844e-05, 6.914138793945312e-05,
6.794929504394531e-05, 6.699562072753906e-05]
```

The Boolean array code is definitely faster. So, despite both versions seemingly consisting of the same steps, the Boolean array approach wins the race. This is because looking up values in an array is faster than looking up values in a hash table. With a hash table, the computer has to perform a hash function on each value to determine where it lives. With an array, no

such computation is needed; a computer knows how to find an index without a hash function.

Note that you can use Boolean arrays even if the range doesn't start at `0`. If, say, the range of integers is from `3000` to `4000`, you can still use a Boolean array in the same way. That is, you simply add `3000` to each index in the array to figure out what integer that index truly represents. So, the index `0` represents the integer `3000` of the Boolean array, and index `456` represents the integer `3456`.

Using a Boolean Array for Counting Sort

We've looked at the Boolean array version of duplicate detection. Let's return to our other application—counting sort—and modify its code to use a Boolean array instead of a hash table:

```
def counting_sort(array):
    set = [False] * 10000
    sorted_array = []

    for value in array:
        set[value] = True

    for number in range(10000):
        if set[number]:
            sorted_array.append(number)

    return sorted_array
```

Here, we initialize a Boolean array designed to represent the integers `0` through `9999`. The code is the same as before, except that our set is housed in a Boolean array rather than a hash table.

My benchmarking results also show that the Boolean array code is faster than its hash table counterpart, although not by as large a margin as with finding duplicates. Here are the results for when these snippets work to sort 9,000 values:

The hash table version yields these speeds:

```
[0.002900791, 0.00256120900000002, 0.00252308399999995,
0.00237707999999999, 0.0023617919999999945]
```

The Boolean array code runs at this speed:

```
[0.0017609160000000013, 0.0016054999999999993, 0.0016088750000000027,
0.0016091659999999952, 0.0015574169999999984]
```

For this example, the Boolean array approach is about 1.5 times faster than its hash table counterpart. It emerges that choosing the right set data structure can make a real difference.

Space-Saving Sets

We've seen how sets, whether in the form of a hash table or a Boolean array, can boost our code's speed in various applications. However, these benefits don't come for free.

Although the brute-force approach of finding duplicates is a slow $O(N^2)$, it does have a saving grace in that it doesn't take up any extra space. That is, the brute-force algorithm doesn't create any additional data structures. However, the set-based approaches create a set that didn't exist before. These sets take up space!

To be precise, the hash table set holds up to N values, as we insert each of the input array values into the hash table. The Boolean array also takes up space, as it holds R values. That is, if our range of data is `0` to `9999`, the Boolean array holds 10,000 Boolean values. And so, the speed that we gained by using a set comes with a trade-off that we must consume extra space. There's no way around the fact that a set will take up memory. That being said, I will now introduce to you a new set data structure that can take up way less space than either a hash table or Boolean array and yet still offer O(1) lookups: bit vectors.

Bit Vectors

Suppose we want to maintain a set of the integers `0, 3, 4, 6`. So far, we've seen two possible ways of doing this.

We have the hash table approach:

```
{0: True, 3: True, 4: True, 6: True}
```

And we also have the Boolean array approach:

```
[True, False, False, True, True, False, True, False]
```

But there's a third approach. What I'm about to show you is one of my favorite tricks in all of computer science. I'll introduce it by revealing one layer at a time.

To start, let's take the previous Boolean array and swap out each `False` for a `0`, and each `True` for a `1`:

```
[1, 0, 0, 1, 1, 0, 1, 0]
```

Recall that with base 2, we can represent any integer using only 0's and 1's. (If you're rusty with base 2, check out the discussion of binary numbers in *Covering All Our Bases*.) With this in mind, we can use base 2 to represent the Boolean array `[1, 0, 0, 1, 1, 0, 1, 0]` using a simple binary number:

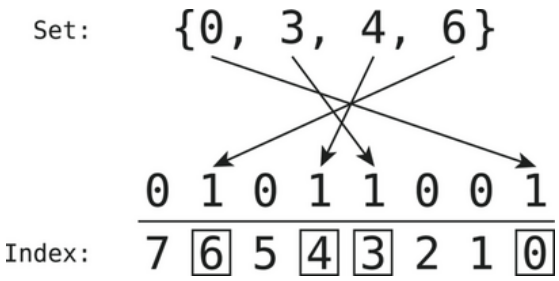
```
10011010
```

We can represent the set of `0, 3, 4, 6` with the single base 2 number `10011010`.

To make things more convenient, we're going to reverse the direction of the way we use bit indexes. That is, with an array, the left-most place is the 0th index. However, if we're using a binary number to represent our data, since the right-most place of any number is the smallest place, it'll be more intuitive if we use the *right-most* place to be the 0th index. And so, the

right-most bit will represent index 0, and the indexes will increase as we move leftward.

Here's a visual of how we're using a binary number to represent our set:



As you can see, because we want to store a `0` from our set, we take the right-most index (the 0th index) and make its digit a `1`. The same goes for all the other numbers in our set. So, the binary number `01011001` represents our set of integers `0, 3, 4, 6`.

In Python, we don't generally interact with integers in binary form. Instead, we work with them as decimal (base 10) numbers. Accordingly, instead of working with the binary number `01011001`, we'd work with its decimal equivalent, which is the integer `89`.

Do you know what this means?

It means that we can store the entire set `0, 3, 4, 6` *using a single Python integer*. Yes, the integer `89` is all we need to represent our set! This has profound implications for how much memory we can save. Our Boolean array had to hold at least seven Boolean values in memory. Likewise, our hash table takes up space with an entire underlying array designed to hold multiple values. But now, we can store the same set inside *a single integer*.

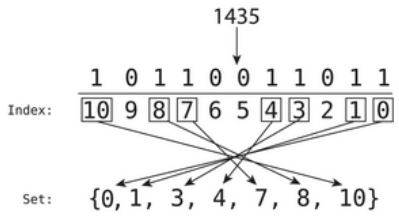
Let's take a look at another example, this time working backward. If we were told that the integer `1435` represented a set of numbers, we could figure out which numbers are in the set by converting `1435` to base 2.

Luckily, we don't have to do this by hand. Python comes with a `bin` function that accepts an integer and converts it to binary:

```
>>> bin(1435)
'0b10110011011'
```

This `'0b10110011011'` is called a *binary string*—it's a Python string object. The starting characters `"0b"` are not part of the binary number and can be ignored. It's there to indicate that the digits that follow it represent a binary number.

We can see that the decimal number `1435` in base 2 is `10110011011`. Armed with this binary number, we can now see which integers are in our set, as shown in the following figure:



The single integer `1435` represents the entire set `0, 1, 3, 4, 7, 8, 10`. This is an incredibly compact way to store a set of 7 values!

This data structure is called a *bit vector*. It goes by other names, too, including bit array, bit set, bit string, and bit map. I call it a bit vector since that sounds the coolest.

In the example, our bit vector is nothing more than a single integer, even though we're treating it as a bona fide data structure. More commonly, though, a bit vector consists of an *array* of integers, as I'll explain in the next section.

Before moving on, though, here's another useful tip: in Python, you can use the expression `0b` to convert a number from binary to decimal. That is, if you type `0b10110011011` in your Python terminal, you'll get a result of `1435`. So, the `bin` keyword converts a number from decimal to binary, and `0b` converts a number from binary to decimal. Try it for yourself!

Bits and Bytes

While bit vectors are cool, there's a catch with storing an entire bit vector inside a single integer: some sets are simply too large to be stored within one integer. Here's why.

As noted in the previous chapter, most computers store their data as binary numbers. This means that although our Python code may deal with integers in base 10, the computer stores these numbers in base 2.

The smallest unit of measurement of computer space is the *bit*—short for *binary digit*—which represents a single `0` or `1`. A number larger than `0` or `1` is represented using multiple bits. For example, the binary number `10000000` (which in decimal happens to be `128`) takes up 8 bits because there are 8 digits in the binary number. Now, generally speaking, a computer reserves *32 bits* in memory to store any integer.

What this means for us is that if we want to use a single integer as a bit vector to store a set of numbers, we'll only have a maximum of 32 bits

available to us. And so, a single integer can only represent a range of numbers from `0` through `31`. Accordingly, we'd have no way to include a `32` or `33`, for example, in our set.

To avoid confusion, note that the Python `bin` command only shows a binary number starting from its left-most `1`. That is, the command `bin(2)` spits out `0b10`, which is the binary number `10`. In truth, though, the integer `2` contains 32 bits. That is, there are another thirty zeroes to the left of the `1` in memory. It's just that Python doesn't bother to show us that for the sake of brevity.

32 Bits or 64 Bits?

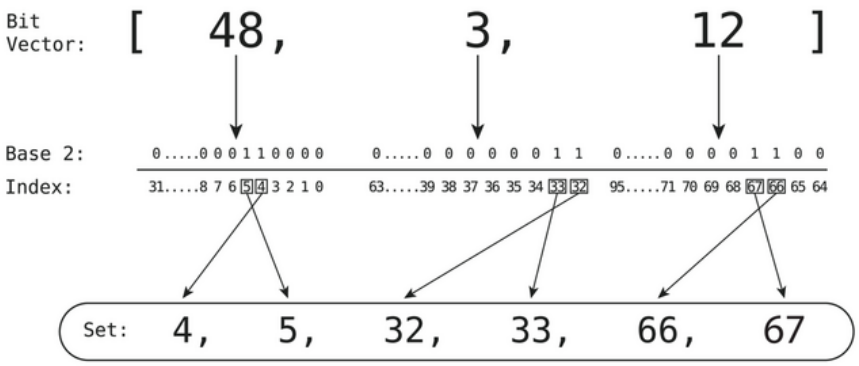
It's not universal that a computer stores an integer using 32 bits. Some computers use alternative schemes, such as storing integers using 64 bits. The number of bits that a computer uses to store an integer is determined by the computer hardware itself.

It's also worth noting that many languages, including Python, pack some extra information into their integers. As such, a Python integer object may be several times larger than a 32-bit machine-level integer.

In any case, a Python integer on many computers is designed to correspond to 32 bits. So, for the purposes of the discussion of this chapter, I'll talk as if a Python integer has 32 bits.

Array-Based Bit Vectors

We have a problem on our hands. How can we use a bit vector to store a set that has numbers that lie in a range larger than `0` to `31`? The answer is that we can use an *array of integers* to serve as our bit vector, as shown in the following illustration:



Here, the bit vector `[48, 3, 12]` represents the set `4, 5, 32, 33, 66, 67`. That is, we use the first integer in the bit vector array (which is `48` in this example) to store values from the range `0` through `31`. The second integer, though, works with an offset of 32, meaning that we use its 32 bits to represent values from the range of `32` through `63`, rather than `0` through `31`. The third integer represents the next range of 32 values, namely, `64` through `95`. And so on.

Although the size of our bit vector has grown, this approach still provides incredible space savings. For example, the array of the following three integers

```
[3254916866, 2148077570, 2164327428]
```

stores this entire set of 18 numbers:

```
[1, 8, 9, 10, 12, 17, 25, 30, 31, 33, 44, 48, 51, 63, 66, 74, 80, 88]
```

In fact, a bit vector of three integers can be used to store up to *96* different values in a set—if every bit were set to `1`. Each additional integer we add to the bit vector can store an additional 32 values since, again, each integer contains 32 bits.

Accessing Individual Bits

We're almost ready to create our own Python implementation of a bit vector. However, there's one teeny, tiny, itty-bitty, little catch.

That is, Python, as well as most coding languages, doesn't give us an easy way to access the different bits within an integer. We are essentially utilizing an integer to store a list of set values, but an integer is not *actually* a list. It's just an integer! Even though you and I can figure out that the decimal integer `2148077570`, when converted to binary, is the number `10000000000010010001000000000010`, how can we write *code* that figures out which bit indexes contain `1` bits?

By contrast, Python arrays allow us to easily look up or change a value at any given index with simple commands like `array[5]` and `array[5] = 1`. But Python doesn't give us such commands for integers. There's no Python function like `44.get_bit_at_index_5` or `44[5]`.

You may be thinking that we can use the aforementioned `bin` command to convert an integer into a binary string, such as

`0b10000000000010010001000000000010`. We can then use a loop to iterate over the string and identify the `1` bits and count which index each one is located at. However, iterating over a string is relatively slow and will neutralize the speed that a bit vector is supposed to offer.

Furthermore, this loop approach only helps for inspecting bits. But we're still at a loss as to how we can *modify* an individual bit. Our bit vector will need to do this since we have to set bits to `1` to represent the different values in our set.

So, how do we quickly read and modify the individual bits of an integer? Well, I have some good news and some bad news and some good news. (Yes, you read that correctly.)

Bit Manipulation

The *good* news is that there *is* a way to quickly access the individual bits that underlie an integer. And so, it *is* possible to implement a fast yet space-saving bit vector. Yay!

The *bad* news, though, is that it's not that straightforward.

But I have some more *good* news: the nonstraightforward techniques unlock an entirely new skill set that is handy to know and also a lot of fun. These dandy techniques are generally referred to as *bit manipulation*. That is, we perform certain special operations to access and modify the individual bits of an integer.

Bitwise Operations

The key to bit manipulation is being able to perform *bitwise operations*, a special set of operations that deal with integers on the bit level. In the following sections, I'll describe each one.

The AND Operation

Open your Python terminal and enter these commands:

```
5 & 6
81 & 103
4 & 8
```

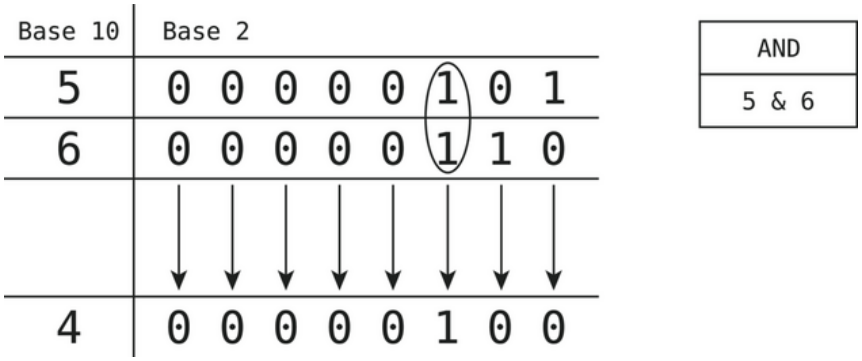
You'll find that you'll get some peculiar results:

```
>>> 5 & 6
4
>>> 81 & 103
65
>>> 4 & 8
0
```

Though this may seem strange at first, it all makes sense when we get down to the bit level.

The `&` operator is the bitwise "AND" operator. (Yes, it's typically written in all caps.) The AND operator takes two numbers, looks at them as binary numbers, and produces a third number whose bits are set to `1` only in digit places where the first two numbers *both* have `1` bits.

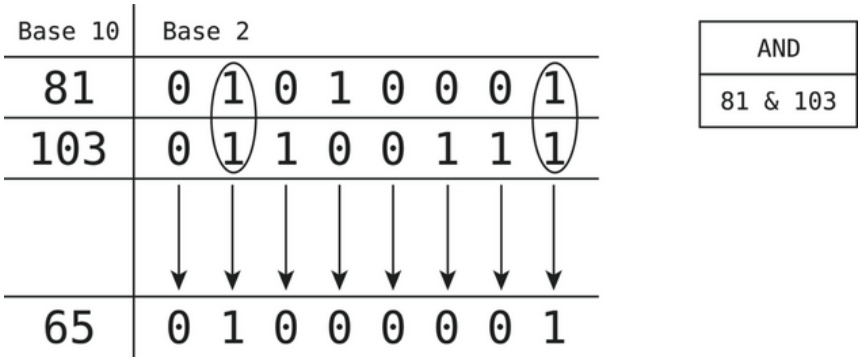
For example, in the following illustration, we AND the numbers `5` and `6`, as shown. (Yes, "AND" can be a verb!)



The integers `5` and `6`, which in binary are `00000101` and `00000110`, respectively, only share a `1` in the third-to-right index. And so, when we AND the two numbers together, the third number's bits are all set to `0` except at that index, where we place a `1` bit. This produces the binary number `00000100`, which in decimal is the number `4`. And that's why `5 & 6` makes `4`.

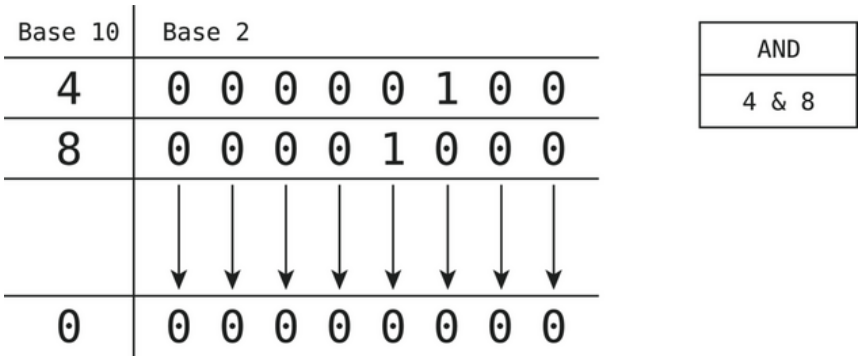
In sum, when we AND two numbers together, the resulting third number will only have `1` bits where *both* of the two original numbers also had `1` bits. This is why it's called "AND"; we need the first number AND the second number to have a `1` bit at the same index.

Let's look at another example. Here's what happens when we AND `81` and `103`:



Here, there are two indexes at which both `81` and `103` have a `1` bit. This yields a result of `01000001`, which in decimal is `65`.

Here's one final example. When we AND `4` and `8`, we get:



Because there isn't a single index in which both `4` and `8` have a `1` bit, `4 & 8` yields `0`.

The following are some key properties of the AND operation, which can all be logically derived from our discussion until this point:

- Whenever we AND a `0` bit with a second bit, the result will always be a `0` bit.
- Whenever we AND a `1` bit with a second bit, the result will be identical to that second bit. (That is, `1 & 0` is `0`, and `1 & 1` is `1`.)
- Whenever we AND two bits that are the same as each other, the result will be the same as those bits. (That is, `0 & 0` is `0`, and `1 & 1` is `1`.)

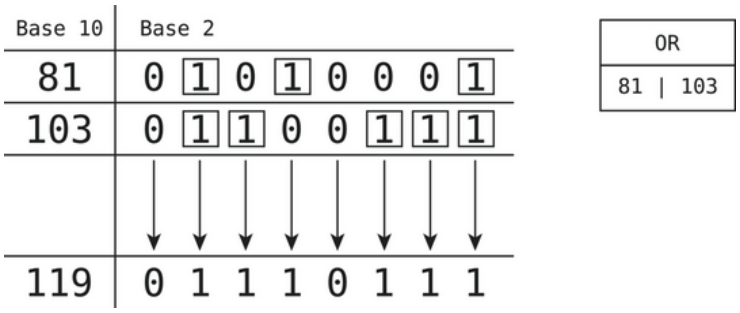
For each bitwise operation, I'll spell out its key properties as I did for the AND operation. Many of these properties may seem obvious, but it will be important to keep them in mind for discussions further on in the chapter. *Trust me*.

The OR Operation

Another major bitwise operation is the OR operation. In Python, we execute this operation with the `|` character, for example, `81 | 103`.

When we OR two numbers together, the resulting third number has `1` bits wherever *either* of the two original numbers had `1` bits. That is, it's enough for only one of the two original numbers to have a `1` bit in order for the third number to have a `1` bit at that same index.

Here's how this plays out for `81 | 103`:



As you can see, the third number gets a `1` bit at any index where *either* of the two original numbers had a `1` bit. Of course, the third number also gets a `1` bit in a spot where *both* original numbers had a `1` bit.

This is why the operation is called "OR": it's enough for either the first number OR the second number to have a `1` bit at a particular index to show up again as a `1` bit in the third number.

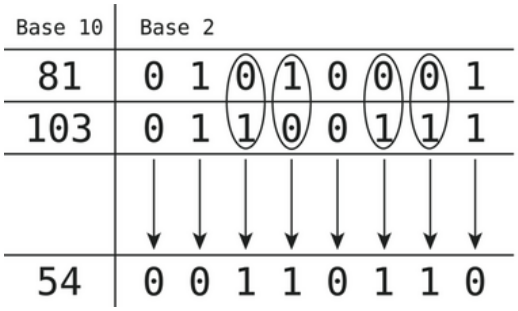
The following are some key properties of the OR operation:

- Whenever we OR a `0` bit with a second bit, the result will be identical to that second bit. (That is, `0 | 0` is `0`, and `0 | 1` is `1`.)
- Whenever we OR a `1` bit with a second bit, the result will be a `1` bit. (That is, that first `1` bit alone will be enough to guarantee that the third number will have a `1` bit.)
- Whenever we OR two of the same bits, the result will be a bit that is the same as those bits. (That is, `0 | 0` is `0`, and `1 | 1` is `1`.)
- Whenever we OR two *opposite bits*, the result will be `1`. (That is, `0 | 1` is `1`, and `1 | 0` is `1`.)

The XOR Operation

The XOR operation is an interesting operation. XOR stands for "Exclusive OR" and is pronounced by many as "ex-or." (A minority of people pronounce it "zor," which certainly sounds cool, but might get you some funny looks.) In Python, we execute XOR using the `^` operator, for example, `81 ^ 103`.

When we XOR two numbers, we produce a third number that contains `1` bits at indexes only where *just one of the two* original numbers had `1` bits. That is, if *both* original numbers have `1` bits at the same index (and certainly if both original numbers had `0` bits at the same index), the third number will have a `0` bit at that index. So, XOR is a little like OR, except that it excludes indexes where *both* of the original numbers contain `1` bits. The following image illustrates the example `81 ^ 103`:



XOR
81^103

Here, there are four indexes where one number has a `0` bit and the other has a `1` bit. Accordingly, these are the four indexes where the third number receives a `1` bit. Again, where the two numbers *both* have a `1` bit, such as the right-most index, the third number receives a `0` bit. This example yields the number `00110110`, which is `54` in decimal.

Another way I like to think about the XOR operation is that it serves as a litmus test that produces `1` bits in each digit place wherever the two integers have *opposite* bits. In other words, it's a great way to see precisely where two integers differ in terms of bits.

The following are some key properties of the XOR operation:

- Whenever we XOR two identical bits, the result will be a `0` bit.
- Whenever we XOR two opposite bits, the result will be a `1` bit.
- Whenever we XOR a `0` bit with a second bit, the result will be that second bit. (That is, `0 ^ 0` is `0`, and `0 ^ 1` is `1`.)
- Whenever we XOR a `1` bit with a second bit, the result will be the *opposite* of that second bit. (That is, `1 ^ 1` is `0`, and `1 ^ 0` is `1`.)

The NOT Operation

Unlike the previous three operations, which use two numbers to produce a third one, the NOT operation uses a *single* number to produce a second number. This operation, which in Python is executed using the `~` operator, is

pretty simple. It produces a new number that is the complete *inverse* of the first number. That is, wherever the first number has a `0` bit, the new number has a `1` bit. Likewise, at each index where the first number has a `1` bit, the new number has a `1` bit. The image shown illustrates the example `~103`:

Base 10	Base 2
103	0 1 1 0 0 1 1 1
152	1 0 0 1 1 0 0 0

NOT
~103

So, `~103`, in theory, should produce `152` since that's the binary inverse of `103`.

That being said, there's a small catch. If we execute `~103` in Python, we get the surprising result of `-104`, and not the `152` we expected.

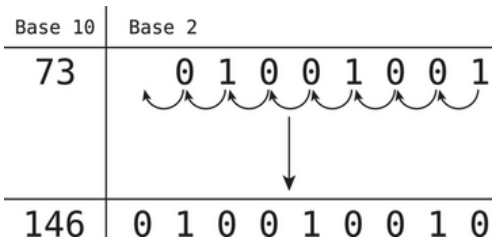
In truth, though, the numbers `-104` and `152` can both be represented using the *same* bit pattern of `10011000`. This is because Python uses a popular numeric system called *two's complement* to represent negative numbers using base 2. (You can read more about two's complement in an article located on the book's web page.[6])

In any case, `~103` *does* indeed produce the bit pattern `10011000` as we expected it would. But `10011000` can translate into either `-104` or `152`, and Python policy is to always return the negative number when we use the `~` operator.

The Shift Operation

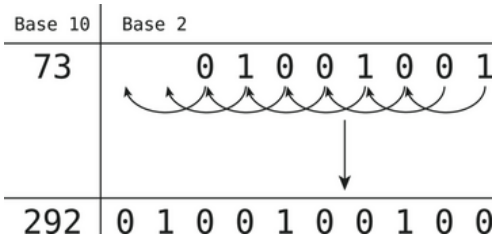
The fifth, and final, bitwise operation is known as *shifting*. Shifting operates on a single number, and we can shift bits either to the left or to the right. Let's look at an example.

The number `73` in binary is `01001001`. We can shift each bit one place to the left with the command `73 << 1`, as shown in the following illustration:



Each bit, whether `1` or `0`, moves leftward by one place. This produces the number `10010010`, which in decimal is `146`. Note that when we shift leftward, the right-most place gets filled in by a `0` bit. There's no digit from the right that we can move into that spot, so we simply place a `0` there.

Now, we can shift by more than one place at a time. If we want to shift each bit leftward by *two* places, for example, we'd write `73 << 2`. What this produces is shown in the image.

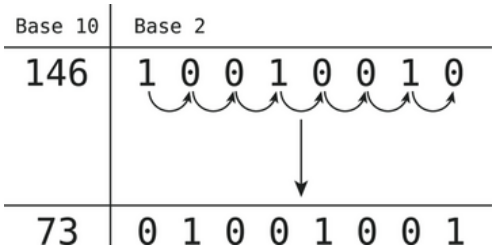


This is the equivalent of executing `73 << 1 << 1`. Note that here, the *two* right-most places get filled in with `0` bits.

What's interesting is that shifting a number leftward by one place, in effect, doubles the number. This is always the case since each `1` bit doubles in value by shifting one place to the left. This makes sense because each digit place represents a number twice as large as the digit place to its right. So, when we shift `73 << 1`, we get `146`, as `146` is double `73`.

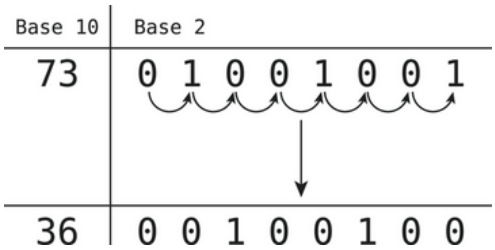
When we shift a number leftward by two places, it's the equivalent of multiplying the number by `4`. When we shift it leftward by three places, we're effectively multiplying the number by `8`. To put this more generally, when we shift a number leftward by K places, we're multiplying the number by 2^K.

We can also shift a number *rightward*. For example, we can shift `146` one place to the right with the command `146 >> 1`, as shown here:



Note that the left-most place gets filled in with a 0. When we shift a number rightward by one place, we are basically halving the number. As is clear from this example, half of `146` is `73`.

Now, let's look at what happens when we shift the number `73` rightward by one place:



Here's something noteworthy: the `73` had a `1` bit in the right-most index. When we shift that bit rightward, it falls into oblivion. So in truth, it doesn't matter what the right-most bit is; whether it's a `0` or a `1`, it goes away.

Now, if the right-most bit of the original number was indeed a `0`, the original number would have been `72`. It turns out that shifting `73 >> 1` and `72 >> 1` *both* yield the same result of `36`.

It emerges that when we shift a number rightward, we divide the number precisely in half *only* when the number is even. That is, with `146 >> 1 = 73`, the number `73` is exactly half of `146`. However, when the number we're shifting is odd, we halve the number and then drop the remainder. That is,

technically speaking, half of `73` is `36.5`. However, when we shift `73` one place to the right, we end up with a rounded-down result of `36`.

Rounding a number down to the nearest integer when dividing is commonly known as *floor* division. And so, we can say that shifting a number rightward by one place halves that number using floor division.

Using Bitwise Operations for Good

Well, that's our motley crew of bitwise operations. At first glance, they can appear kind of random and not particularly useful. However, these bitwise operations can be *incredibly* useful. In fact, all math operations that a computer performs rely on these operations under the hood! Let's look at one example.

Adding Binary Numbers

If we were to use pencil and paper to add two binary numbers together, such as `5 + 2`, it would look like this:

```
5 →   0101
2 → + 0010
    ------
7 →   0111
```

This example is pretty simple in that we didn't have to carry any numbers. If we had to carry numbers, as in the example of `5 + 1`, it would look like this:

```
        1 ←carry the 1
5 →   0101
1 → + 0001
    ------
6 →   0110
```

Carrying works the same way as with base 10. It's just that whenever we add two `1` bits together, we have to carry that `1` bit to the next place to the left. (With base `10`, we carry a `1` when the sum of two digits is greater than `9`.)

Now, here's the interesting thing. At the most primitive machine level, a computer *does not have a `+` operation*. Just as a computer at the machine level only deals with binary numbers (rather than decimal numbers), it likewise *only deals with bitwise operations*. This means that a math operation as simple as adding two numbers can only be done by a computer by using bitwise operations.

But how, exactly, do we use bitwise operations to add two numbers together?

This answer is more straightforward in cases where we don't need to carry. Take a look again at the previous diagram showing the bit addition example of `5 + 2`. No carrying is needed there, as there's no index where both numbers have a `1` bit. Because of this, conveniently enough, we can perform the addition by simply XORing the `5` and `2`. This may seem like a cool coincidence at first, but it makes sense after thinking about it a bit.

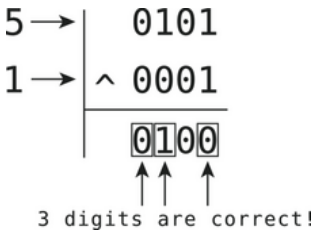
Take a good look at the `5 + 2` visual. The result is, in fact, the same as XORing the `5` and `2`. The reason this works is that when we add two `0` bits together, we want to get `0`. Indeed, XORing does this. Similarly, we need it to be that when we add a `0` bit and a `1` bit, we get a resulting `1` bit. XORing does this as well. Sweet!

The tricky part, though, is when we have a case where both numbers have a `1` bit at the same index. Our goal in that case is to have a result of `0` at that index, which XORing takes care of as well. However, we also need to take care of carrying a `1` to the next place over to the left, which XORing does *not* do automatically.

That is, if we simply XOR two numbers without performing any extra steps, the necessary carrying will not take place, and we'll end up with an incorrect result. Let's look at this visually for the case of `5 + 1`. If all we do is XOR the `5` and `1` together, we end up with:

```
5 →|  0101
1 →| ^ 0001
   |-------
4 →|  0100
```

This would indicate that the sum of `5` and `1` is `4`, which is clearly incorrect. However, although this result is incorrect, it's headed in the right direction. The result of the XOR operation does, after all, make it so that three out of the four digits of our result *are*, in fact, correct, as shown in the illustration.



Here, we get a result of `0100`, while the true sum we want is `0110`.

So, it turns out that XORing the two numbers gets us most of the way to the desired solution. It produces what I call the "sum-without-carry." That is, it produces a sum that is accurate if we don't have to carry any digits. And even if we *do* need to carry, the sum-without-carry is *almost* accurate. The only thing we're missing is the carrying of numbers.

Luckily, we can perform the carrying with some additional bitwise operations.

Carrying

The general strategy for carrying goes like this. If we could, somehow, compute what I call the "carry number," we could add it to the sum-without-carry to get the correct sum. Here's what I mean.

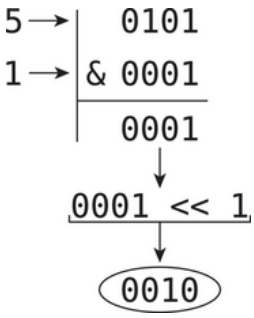
In the previous example of `5 + 1`, our sum-without-carry is `0100`. This isn't the correct sum, since we failed to carry a `1` bit over to the second-to-right-most digit place. Now, this `1` bit we failed to carry represents the binary number `0010`. Put another way, a `1` bit in the second-to-right-most digit place represents the decimal number `2`, which in binary is `0010`.

And so, the number `0010`—which I call the "carry number"—is what we're missing from our total sum.

It follows that if we were to *add the carry number to the sum-without-carry*, we'd get the true sum. That's what carrying is: we're adding a carry number to the other numbers we're working with. Indeed, when we add the carry number `0010` to the sum-without-carry of `0100`, we get `0110`. This, in decimal, is `6`, which is the correct answer to `5 + 1`.

For the computer to add the carry number to the sum-without-carry, though, the computer first needs to figure out what the carry number is. Fortunately, we can do this with a clever little trick.

This trick relies on the following observation: if we AND our two numbers, and then shift the result leftward by one place, the result will be the number we're supposed to have carried, as shown in the illustration.

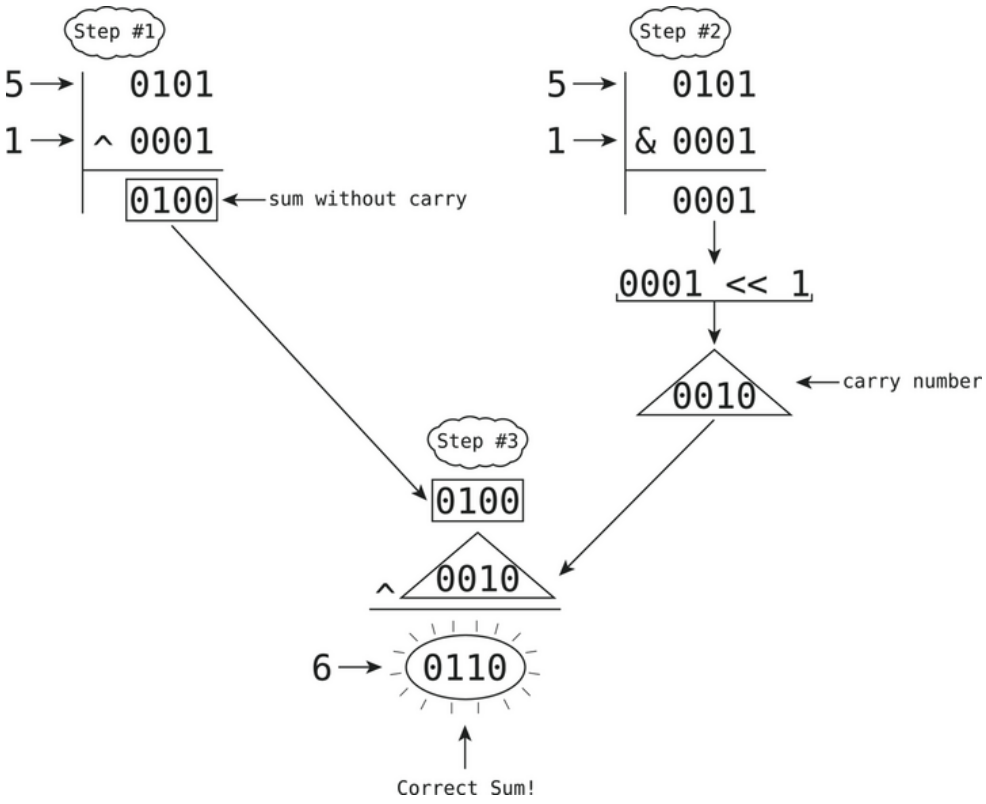


This result of `0010` is indeed the carry number we're looking for. Here's the reason why ANDing with left shifting successfully computes the carry number:

A carry number is only produced when we add two `1` bits together. The AND operation does this; it produces a third number that shows us exactly where the original two numbers shared `1` bits. However, we need to, well, *carry* that `1` bit over to the next digit place to the left. We accomplish this, though, with the left shift!

Okay, here's where we're at so far:

We have the XOR technique that computes the sum without carrying. We now also have the AND-with-shift to produce the carry number. Next, we need to add the sum-without-carry and carry number together. In our example of `5 + 1`, we can do this by XORing the sum-without-carry and carry number together. Here's how this all comes together:



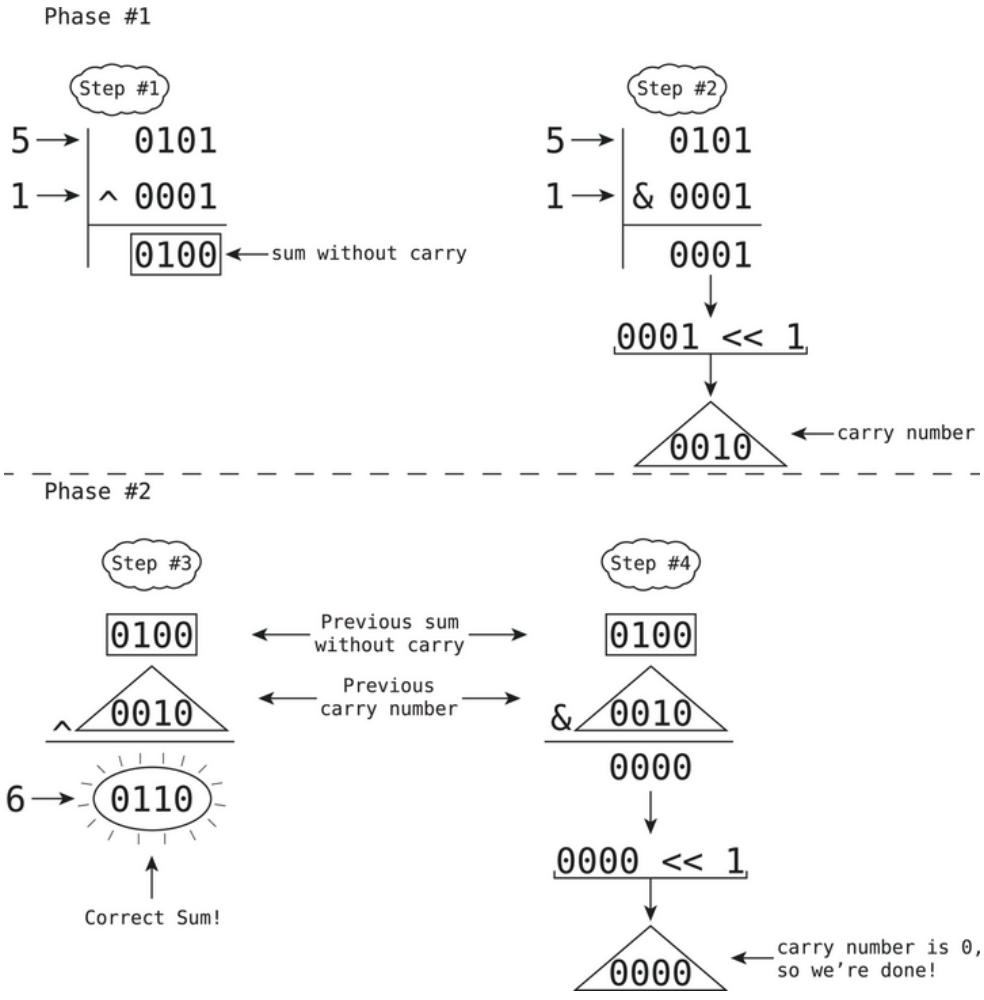
In the previous diagram, Step #1 computes the sum-without-carry (using XOR). Step #2 computes the carry number (using AND with left shift). Step #3 then "adds" the sum-without-carry and the carry number with another XOR command. This gives us the final, and accurate sum of `6`!

You may be wondering how Step #3 successfully added the sum-without-carry and carry number, as earlier you saw that XORing isn't itself the same as adding. Rather, XORing only computes the sum-without-carry!

In fact, you're right. We happened to get lucky with the example of `5 + 1` since in Step #3, when we XORed `0100` and `0010`, there *was* no carry

number! And so, we get the correct answer because the sum-without-carry is indeed correct in cases where no carrying is needed.

In truth, there's really a Step #4 to the process for adding binary numbers. Step #3 XORed our sum-without-carry with the carry number to produce a new sum-without-carry. Step #4 then uses AND with left-shift again to compute a new *carry number* as shown here:

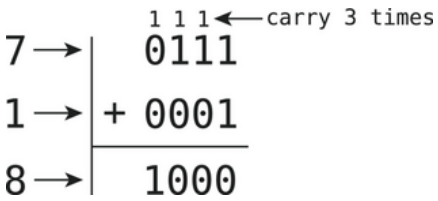


In this example, Step #4 produces `0000`, which means that there's no carry number. (There's only a carry number if there's at least one `1` bit. This means that our algorithm is complete.

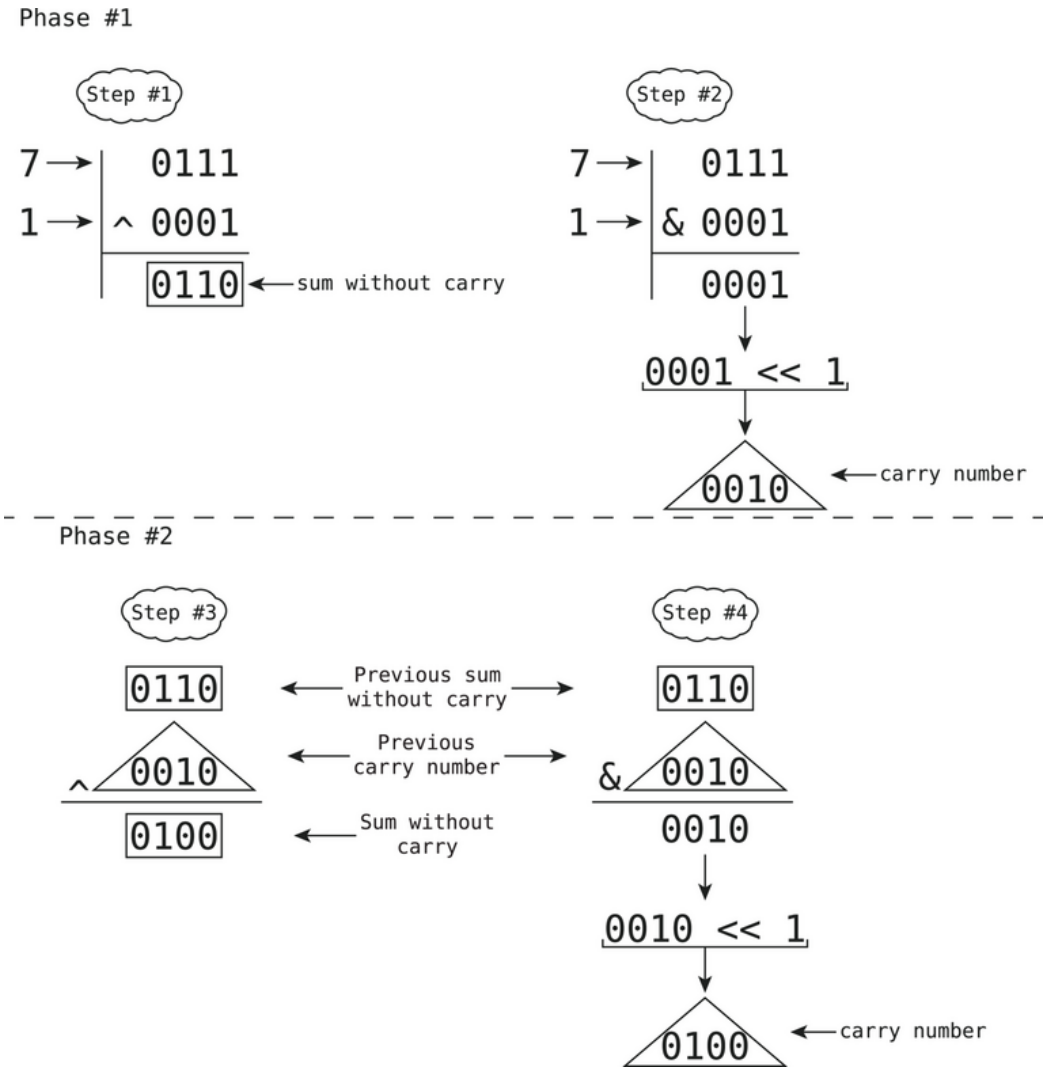
However, in cases when Step #4 *does* produce a carry number, we need to loop and keep repeating the same steps of XORing and also ANDing plus

left-shifting. We do this until we end up with no carry number. Fortunately for us, we will reach such a point in all cases, at least eventually.

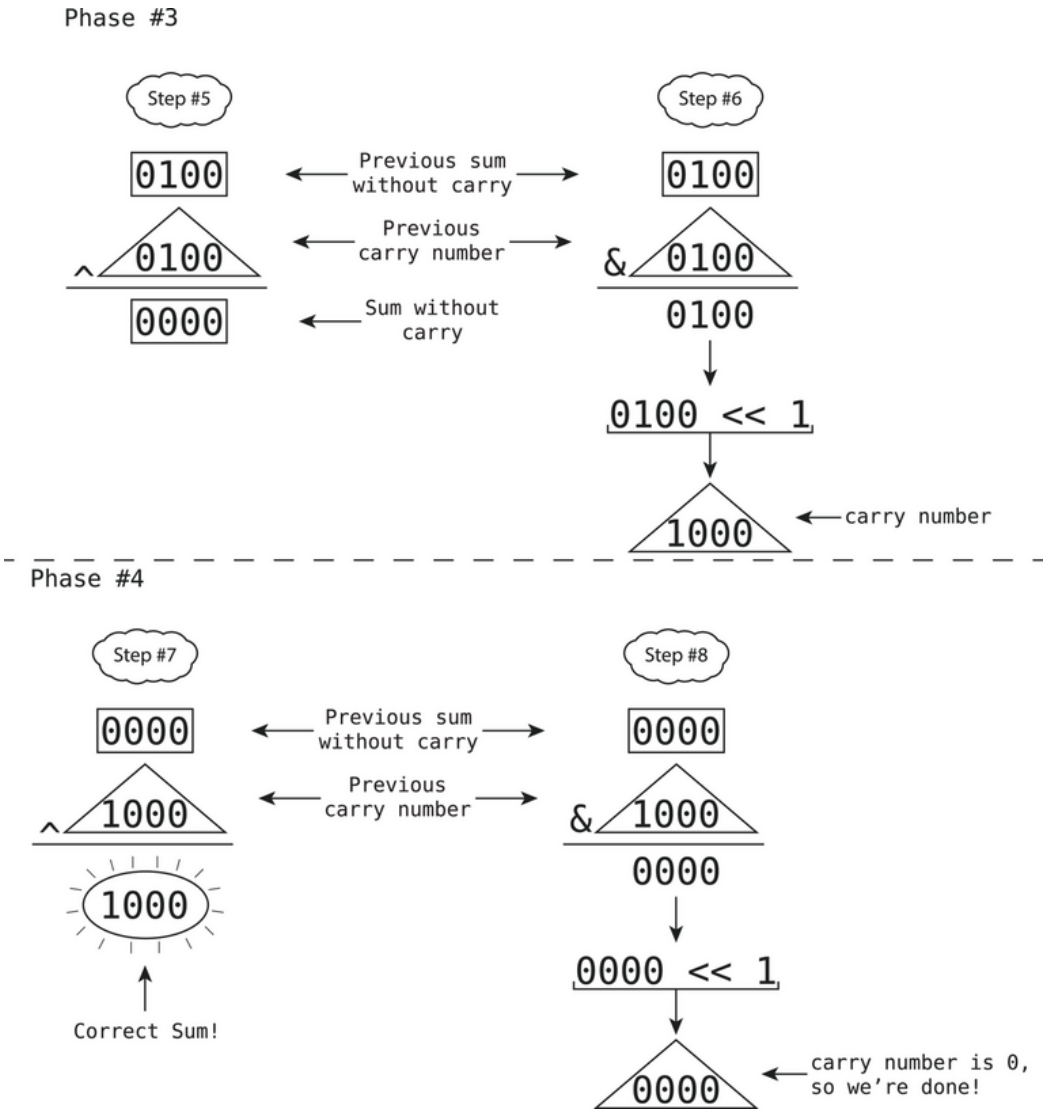
Here's an example of what I mean. Suppose we want to add `7 + 1`. If we did this by hand, we'd get:



As you can see, if we perform good old pencil-and-paper addition, we end up having to carry a `1` *three* times. Performing this computation will be a bit of a doozy. In this scenario, when we perform Step #4, we do end up with a carry number:



This means that we need to keep repeating the same process until we're left without a carry number. Here's what this looks like:



Because Step #4 ends with a carry number, Step #5 XORs the *new* sum-without-carry with the *new* carry number to produce yet another sum-without-carry. Step #6 then computes a new carry number that is to be added with the latest sum-without-carry.

Step #6 demonstrates that we still have to carry another number, so we loop again and perform Steps #7 and #8. Step #8 yields no carry number, and the binary addition is complete.

Whew!

Code Implementation: Adding Binary Numbers

Fortunately, the code for adding binary numbers is short and to the point:

```
def add(first_number, second_number):
    while second_number != 0:
        sum_without_carry = first_number ^ second_number
        carry_number = (first_number & second_number) << 1
        first_number = sum_without_carry
        second_number = carry_number

    return first_number
```

We run a loop where, in each iteration, we first XOR the two numbers we're adding to produce a `sum_without_carry`. We then AND the two numbers and shift one place leftward to produce a `carry_number`.

We then want to add the `sum_without_carry` to the `carry_number`, so we update the `first_number` to now be the `sum_without_carry` and the `second_number` to now be the `carry_number` and repeat the loop again.

The loop terminates once the `second_number`, which is equivalent to the `carry_number` (after the first round of the loop is complete), is `0`. At that point, we return the `first_number`, which is equivalent to the `sum_without_carry`.

You've now seen how bitwise operators are a lot more useful than they look. Indeed, addition is not the only piece of arithmetic that a computer can execute using bitwise operators. Virtually *all* math that a computer does uses bitwise operators under the hood. Furthermore, computer scientists and mathematicians have discovered various other cool tricks that bitwise operators can pull off. There's no room for us to discuss all of them here, but hopefully the process of adding two numbers gives you a taste of this.

Now, let's go back to where we began and use bitwise manipulation to build our bit vector.

Bit Masks: The Key to Zeroing in on a Bit

The key to making our bit vector work is to gain the ability to access and modify individual bits of an integer. As a reminder, our goal with a bit vector is to do things like represent the set `0, 3, 4, 6` using a single integer such as `89`. Again, the way this works is that `89` in binary is `01011001` and, therefore, has `1` bits at indexes `0`, `3`, `4`, and `6` (with index 0 being the right-most bit and the indexes increasing going leftward). Although you and I can look at the binary number `01011001` and see which indexes contain `1` bits, we need a way to write code so that the *computer* can do this as well.

Here's another way to think about this problem. I once read a true story about a child genius who, despite his genius—or more accurately, *because* of his genius—couldn't learn to read. The problem was that when he saw a page of words, his mind took in all the letters and words simultaneously. As such, he could never focus on just a single word.

The same applies to our case. Even if we enter into our Python terminal the binary string `0b01011001`, the computer will immediately spit out `89` because, like the genius, the computer is looking at the entire number as a whole. We need to somehow get the computer to identify a single bit within that number.

Now, a specialist *did* end up finding a way to teach the child prodigy to read, and the child eventually grew up and went on to become a highly successful and prolific educator himself. What was the specialist's trick?

The specialist figured out that if they covered an entire book page *except for one word*, they could get the child to focus on that word alone. By blocking the child's view of all the other distracting words, the child was able to take in one word at a time. To read the next word, they revealed the next word only after blocking the previous word.

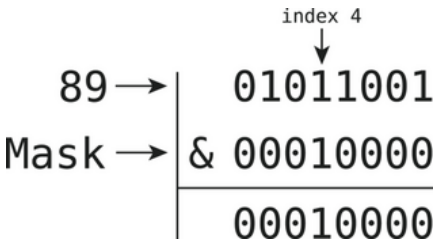
And so, we're going to use that same trick to get the computer to access individual bits.

The general approach for accessing individual bits of an integer is by using something called a *mask* (or *bit mask*). A mask is, in fact, an integer, but it's an integer whose bits are cleverly set to allow us to focus on a specific part of *another* integer. Let's take a look at an example using the integer `89`. Again, in base 2, this is `01011001`.

Suppose we want to check whether the bit at index `4` is a `0` bit or a `1` bit. You and I can see that it's a `1` bit, but the computer sees the entire number of `89`. Here's how we get the computer to focus only on the bit at index `4`.

We create a new integer—the mask—whose bits are all set to `0` *except* for the bit at index `4`, which is set to `1`. This is the integer `00010000`. In decimal, this happens to be `16`, but we don't care about that. What we care about is that the only bit set to `1` is the bit at index `4`, and that the rest are `0` bits. This integer, `00010000`, is our mask.

Next, we AND our integer of `89` with this mask:

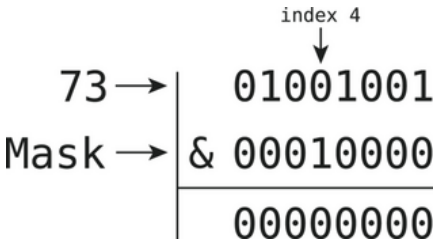


The result is a number whose bits are all `0` except for the bit at index `4`. That is, wherever our mask contains a `0` bit, we "cover" the corresponding bit of the `89` since when we AND each of the `89`'s other bits with the mask's corresponding `0` bit, the resulting bit will always be `0`.

However, our mask has a `1` bit at index 4. This will enable us to see what bit lies in the `89`'s index `4` since when we AND the bit at the `89`'s index `4` with

the mask's 1 bit, the result will always be the same as the 89's index 4 bit. As you've learned, a key property of AND is that whenever we AND a 1 bit with another bit, the result is always that other bit.

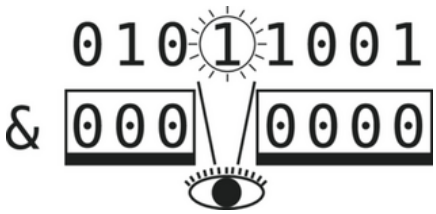
Now, suppose we want to identify the index 4 bit of the integer 73. This bit is a 0. Therefore, when we AND the 73 with our same mask, 00010000, we get:



It comes out that when we AND any integer with a mask, if the resulting integer is 0 (which is a slew of 0 bits), we know that the bit we're inspecting is a 0 bit. However, if the resulting integer is anything *other* than a 0, we know that the bit we're inspecting is a 1 bit.

When we used the mask on the integer 89, the result was 00010000. In decimal, this happens to be 16, but the thing we care about is simply whether the result is 0 or not. Because it's 16, and 16 is most certainly not 0, we know that the bit we're inspecting is a 1 bit.

This is essentially the same approach the specialist used with the genius child. We're using the mask's 0 bits to block the computer's view of the 89's other bits so that it can focus solely on the bit at index 4, as shown here:



We leave an opening—by way of a 1 bit at index 4—to focus and see which bit lies at the 89's index 4. Without the mask, all the other bits get in the way

because the computer naturally sees all the bits together as a whole without being able to isolate a single bit. The mask, though, is what allows us to isolate a single bit.

In short, the `1` bit of the mask at index `4` serves as the opening for the computer to see what lies behind it. This is because when we AND a `1` bit with another bit, the result will always be that other bit. So, if the resulting integer is `0`, we know that our bit in question is also `0`. If the result is anything other than `0`, then our bit in question must be a `1`.

The following code creates and uses this mask:

```
mask = 0b00010000
mask & 89
```

The output of this code is `16`, which means that there's a `1` bit at index `4`. If there were a `0` bit at index `4`, the result would have been `0`.

A Get-Bit Function

As we've seen, with a code expression like `0b00010000 & 89`, we can use a mask to get the computer to reveal an individual bit at any index of an integer. Let's now use this idea to write a *function* that checks a bit at a given index of a particular integer.

That is, we'll create a function, `get_bit`, which will accept two parameters. The first parameter is our integer, and the second parameter is the index of the bit we want to inspect. Let's start writing it:

```
def read_bit(integer, index):
    mask =  # what code goes here???
    return (mask & integer) != 0
```

This code is only half-baked, but the final line of code makes sense. The final line ANDs the `mask` and `integer` together and then returns `False` if the

result is `0` or returns `True` if the result is any integer other than `0`. But we haven't yet figured out how to write code to produce the `mask` itself.

To solve this, we'll once again use the power of bitwise operations. Specifically, we'll use the shift operator to solve this task! I'll present the code first, and then explain it:

```
def read_bit(integer, index):
    mask = 1 << index
    return (mask & integer) != 0
```

To create our `mask`, we start with an integer `1`. This is equivalent to `00000001`, where the `1` bit is at index `0` of the integer. If we want the `1` bit of our `mask` to be at index `4` instead, all we have to do is simply *shift the `1` bit four places to the left*. This produces our desired `mask` of `00010000`. In other words, we can take the integer `1` and left-shift it by whatever number `index` is. Cool!

Okay, we're making progress. We've successfully written code to read individual bits from an integer. Next up, let's figure out how to *change* bits of an integer, that is, flipping a bit from `0` to `1` or vice versa.

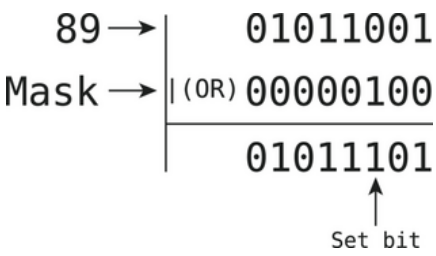
In bitwise manipulation jargon, the term for flipping a bit from `0` to `1` is called *setting* the bit. And the term for flipping a bit from `1` to `0` is called *clearing* a bit. Let's now write the code for each of these operations.

A Set-Bit Function

Once again, we're going to use a mask to help us achieve our task at hand. As with the mask we used to read a bit, our mask will be a series of zeroes except for the index whose bit we want to set. This index will contain the only `1` bit.

To *set* our desired bit (that is, to make the bit `1`), we simply OR the mask with our integer. Let's see how this works.

Suppose our integer is `89`, which in binary is `01011001`. To set its index `2` bit, we create a mask of zeroes with the index `2` bit set to `1`, which is `00000100`. We then OR the mask with the `01011001`, which yields:



This result is the same as the `89`, except that its bit at index `2` is now a `1` bit. I'll call this bit the "setting bit."

The trick here is based on one of the key properties of OR. Whenever we OR any bit with `1`, the result will always be a `1`. So, because we OR the setting bit with a `1`, the bit will now be a `1` bit whether it was a `1` bit before or not.

At the same time, our mask ensures that we don't modify any of the other bits. This works because the rest of our mask (other than the setting bit) consists of `0` bits. Another key property of OR is that whenever we OR `0` with another bit, the result will be that other bit. So, we leave all the other bits unchanged.

We can implement this as a function:

```
def set_bit(integer, index):
    mask = 1 << index
    return integer | mask
```

That is, we first create the `mask`, and then OR the `integer` with that `mask`.

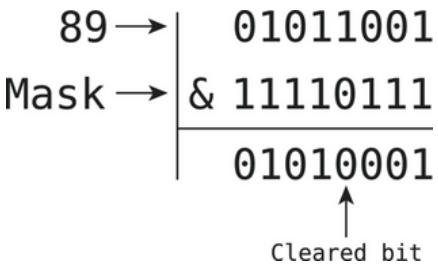
Note that we're not modifying the original `integer` since that isn't something that Python does. Instead, we're returning a new integer that is the result of ORing the original `integer` with our `mask`. The same goes for the methods that follow.

A Clear-Bit Function

If you can believe it, clearing a bit is even more fun than setting a bit. To clear a bit, we're going to create a different type of mask than before. That is, previously, our masks were a bunch of zeroes with a single `1` bit. But now, we'll do the opposite, as our mask will be a bunch of `1` bits, with only a single `0` bit.

Again, let's work with the example integer `89`. Suppose we want to clear the bit at index `3`. To do this, we create a mask of `1` bits except for the bit at index `3`, which is a `0` bit. Our mask, then, is `11110111`.

We then AND the mask together with the `89`, which gives us this:



This result is exactly what we want; it's an integer that's identical to `89` except that its bit at index `3` was flipped to `0`. I'll call this bit the "clearing bit."

The reason this works is that we're ANDing the clearing bit with `0`. And as we've seen, whenever we AND any bit with `0`, the result will always be `0`.

At the same time, our mask ensures that we don't change any other bits. We accomplish this by ANDing all the other bits with `1`. We've seen that one of the key properties of AND is that when we AND `1` with any bit, the result will be that other bit, unchanged.

In sum, to clear a bit, we AND the clearing bit with `0`, while ANDing all the other bits with `1` to leave them as they are.

The nerd inside of you is loving this, I know. But now let's get to the fun part—creating the mask.

Our goal is to create a mask that consists of `1` bits except for a `0` bit at the desired index. Now, it can be tricky to kick things off by setting an integer that contains only `1` bits because it's not always clear as to what Python integer we should use. Sure, the integer `255` is a series of eight `1` bits, but if the size of a computer's integers is, say, 32 bits, there will be 24 `0` bits on the left-most side of the integer. And we don't want that, since we need *all* of our bits (save for one) to be `1`. Furthermore, many computers use 64-bit integers, so we can't always know with certainty what integer we should use to initialize our mask.

However, this is where the NOT operation is our friend.

Suppose we want to clear the bit at index `4`. This means that we want a mask along the lines of `11110111`. To accomplish this, we'll start by creating the *opposite* mask, `00001000`, which we've already seen how to do earlier. (That is, we'll take the integer `1` and shift it leftward four places.) Then, to produce our desired mask, we simply NOT the mask to invert its bits! Thus, `00001000` becomes `11110111`, which is precisely the mask we want.

This approach works no matter the number of bits in an integer. Whether there are 10 `0` bits trailing on the left-hand side of our initial mask, or 20 such trailing `0` bits, all these bits will now be flipped to `1` bits.

In code, we can create a function that executes this strategy:

```
def clear_bit(integer, index):
    mask = ~(1 << index)
    return integer & mask
```

That is, to create our `mask`, we first shift a `1` bit leftward and then NOT the result with the `~` operator. We then return the result of ANDing this `mask`

with our `integer`.

A Toggle Bit Function

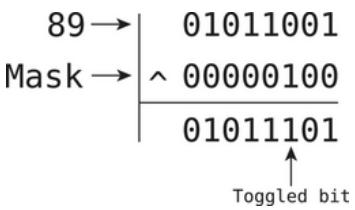
We're on a roll, so let's create another classic bit vector function. To *toggle* a bit is to flip it to the opposite of what it currently is. So, if the bit is currently a `0`, toggling it will set it to `1`. Likewise, if the bit is currently a `1`, toggling it will clear it to `0`. I'm going to call the bit we're interested in toggling the "toggle bit."

Technically, we could accomplish this by using an `if` statement in conjunction with our `get_bit`, `set_bit`, and `clear_bit` functions. That is, we can use `get_bit` to check whether the toggle bit is a `0` or `1`. If it's a `0`, we'll run the integer through the `set_bit` function, and if the toggle bit is a `1`, we'll run the integer through the `clear_bit` function.

While this approach will certainly do the trick, there is a much more concise—and nerdier—method.

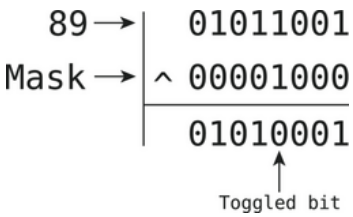
To toggle a bit, we first create a mask of zeroes, except for a `1` bit that lives at the index of our toggle bit. Then, we XOR the original integer with our mask. And that's it! Let's look at how this all plays out.

Suppose we want to toggle the bit at index `2` of the integer `89`. To do this, we XOR the `89` with our mask as shown here:



Boom! The bit at index `2` gets toggled from a `0` to a `1`.

Now, let's toggle index `3` from the `89`:



Here, the bit at index `3` gets toggled from a `1` to a `0`.

The reason this all works is because of the key properties of XOR. That is, if the toggle bit is currently a `0`, XORing it with `1` produces a `1` bit. On the other hand, if our toggle bit is a `1`, XORing it with `1` produces a `0` bit.

And so, when we XOR our toggle bit with `1`, the result will be the opposite of the toggle bit.

At the same time, we leave all the other bits unchanged by XORing all of them with `0`. A `0` bit XORred with `0` produces `0`, while a `1` bit XORred with `0` produces `1`. And so, we can code up a toggle function, keeping it short and sweet, like this:

```
def toggle_bit(integer, index):
    mask = 1 << index
    return integer ^ mask
```

Now that we have a general approach for getting, setting, clearing, and toggling bits, we're finally—at long last—ready to implement our own bit vector.

Code Implementation: Bit Vector

The code we wrote in the previous section is almost everything we need to implement our bit vector; there's only one missing piece. As I mentioned earlier, a single integer can only hold 32 bits, and it can therefore only store a set of values that range from `0` to `31`. To store additional values, we'll need an *array* of integers.

Following is a Python implementation of a bit vector that uses an array of integers to store its values. The scheme goes like this: the first integer represents values in the range of `0` to `31`. The second integer represents values in the range of `32` to `63`. The third integer represents values in the range of `64` to `95`, and so on. Each next integer within the array will store the next set of 32 values.

First, I'll show you the code, and then we'll walk through it line by line:

```
class BitVector:
    def __init__(self, range_of_bits):
        self.range_of_bits = range_of_bits

        integers_length = range_of_bits // 32
        if range_of_bits % 32 != 0:
            integers_length += 1
        self.integers = [0] * integers_length

    def read_bit(self, index):
        integer_index = index // 32
        bit_index = index % 32
        mask = 1 << bit_index
        return (mask & self.integers[integer_index]) != 0

    def set_bit(self, index):
        integer_index = index // 32
        bit_index = index % 32
        mask = 1 << bit_index
        self.integers[integer_index] |= mask

    def clear_bit(self, index):
        integer_index = index // 32
        bit_index = index % 32
        mask = ~(1 << bit_index)
        self.integers[integer_index] &= mask

    def toggle_bit(self, index):
        integer_index = index // 32
        bit_index = index % 32
        mask = 1 << bit_index
```

```
            self.integers[integer_index] ^= mask

        def values(self):
            set = []
            for number in range(0, self.range_of_bits):
                if self.read_bit(number):
                    set.append(number)

            return set
```

When creating a new bit vector, the user is expected to include as a parameter the number of expected values. That is, if we expect to store a set whose values range from `0` to `255`, we'll initialize the bit vector with code like this:

```
bv = BitVector(256)
```

The underlying data structure behind the bit vector is the `self.integers` array, which is, you guessed it, an array of integers. When the bit vector is first initialized, the constructor creates that array and then fills it with the appropriate number of zeroes. If we determine that `self.integers` needs to hold five integers, this means the array will start out as: `[0, 0, 0, 0, 0]`.

To compute how many `0`s we need to fill our array, our constructor first makes the assumption that each integer contains 32 bits (each of which will be either a `0` or a `1`). If our bit vector stores values within a range of `0` to `255`, this will require 256 bits. Because each integer in our `self.integers` array can store 32 of these bits, we'll need our array to hold eight integers in total since `256 // 32 = 8`.

And so, the most important code of our constructor sets this all up with:

```
integers_length = range_of_bits // 32

self.integers = [0] * integers_length
```

(Note that we could have also performed the same calculation with `integers_length = range_of_bits >> 5` since shifting rightward by `5` is the equivalent of dividing by `32`! But I used the regular division operator to make the code easier to understand.)

In any case, the `range_of_bits // 32` calculation works perfectly if the `range_of_bits` is divisible by `32`. But let's say our scenario required 1,000 bits. If we divide 1,000 by 32, we get 31.25. This would mean that our `data.integers` array would have to hold 31.25 integers, but there isn't such a thing as a quarter of an integer!

This means that, practically speaking, we'll need 32 integers. To account for this, our code checks to see whether there's a remainder in `range_of_bits % 32`, and if there is, we add an extra integer inside the `self.integers` array to be able to store those extra bits. And so, we have the following code:

```
if range_of_bits % 32 != 0:
    integers_length += 1
```

Getting back again to the final line of our constructor, `self.integers = [0] * integers_length` is what fills the `self.integers` array with zeroes. So, if `integers_length` is `8`, for example, the `self.integers` will end up being `[0, 0, 0, 0, 0, 0, 0, 0]`. *This simple array is the heart of the bit vector*.

The `read_bit` function is almost identical to how we wrote it earlier, except that before, we simply passed a single integer to read one of its bits. Now, however, we have to first identify which integer within `self.integers` we need to access. Once we grab the correct integer, we can read a bit from it.

The first half of the `read_bit` function goes like this:

```
def read_bit(self, index):
    integer_index = index // 32
    bit_index = index % 32
    # ...
```

To find the appropriate bit within the appropriate integer, we use division, getting both the quotient and the remainder. Because each integer contains 32 bits, we divide the `index` we're reading from by 32. For example, if we want to inspect the bit corresponding to the value `64`, we divide `64` by `32`. This gives us a quotient of `2`, and a remainder of `0`. This means that the bit representing `64` can be found in the integer at index 2 (of `self.integers`), and specifically at zeroth bit index within that integer. That is, it's the first bit of the third integer.

Similarly, if we desired to access the bit corresponding to the value `65`, we once again divide this number by `32`. This time, we get a quotient of `2` but a remainder of `1`. This indicates that the `65` bit is found at index `2` (of `self.integers`), but this time the specific bit index *within* this integer is `1`. In other words, the `65` bit is the *second* bit of the *third* integer.

So, our `read_bit` method calculates both an `integer_index` and a `bit_index`. The `integer_index` tells us the index of the integer within `self.integers` we need to inspect. The `bit_index` tells us which bit *within* that single integer we want to read.

The final two lines of `read_bit` work as our original `read_bit` method of a single integer did. It's just that now we use the `bit_index` to compute the mask and then AND it with the desired integer from `self.integers`:

```
mask = 1 << bit_index
return (mask & self.integers[integer_index]) != 0
```

If the result is `False`, it means that the bit we're reading is `0`, and if the result is `True`, the bit is `1`.

Let's now jump into the `set_bit`, `clear_bit`, and `toggle_bit` methods. The methods begin by computing the `integer_index` and `bit_index` in the same way that the `read_bit` method did. Any time we modify a bit, we do so by

updating the integer in which that bit lives. With the `set_bit` method, we can't merely return a new integer as our original `set_bit` method did. Instead, we now modify the target integer (which is `self.integers[integer_index]`) with the following code:

```
self.integers[integer_index] |= mask
```

The `|=` operator works much like the `+=` operator. That is, just as `x += 1` is equivalent to `x = x + 1`, similarly, `x |= mask` is equivalent to `x = x | mask`.

Along the same lines, the `clear_bit` method updates the target integer with `self.integers[integer_index] &= mask`, and the `toggle_bit` method updates the target integer using `self.integers[integer_index] ^= mask`.

I also included a `values` method in our `BitVector` class. This method returns an array containing the set of values that our bit vector represents. That is, if our bit vector's `self.integers` is `[89]` in order to represent the set `0, 3, 4, 6`, then `values` will return the array `[0, 3, 4, 6]`.

And that's it!

Using Our Bit Vector

Now that we have a working bit vector, let's use it to serve as our set for the algorithms we looked at earlier: finding duplicates and counting sort.

In the following code, I've rewritten our finding duplicates algorithm by using a bit vector instead of a hash table or Boolean array:

```
import bit_vector


def has_duplicates(array):
    set = bit_vector.BitVector(1024)

    for item in array:
```

```
            if set.read_bit(item):
                return True
            else:
                set.set_bit(item)

        return False
```

I set the bit vector to hold values in the range of `0` through `1023` in this example, assuming that our `array` will only hold values in that range. Obviously, if you know that `array` will hold a different range of values, you'd adjust this accordingly.

We can also now rewrite our counting sort algorithm using a bit vector:

```
import bit_vector


def counting_sort(array):
    set = bit_vector.BitVector(10000)

    for value in array:
        set.set_bit(value)

    return set.values()
```

Interestingly, when I benchmark these snippets, they run a little slower than the Boolean array and hash table implementations. However, the advantage of bit vectors is the *space savings* they offer.

Let's see how much space bit vectors save us.

Benchmarking Space

We can benchmark space just as we can time, and thankfully, it's easy to do with Python. Here's how we can use Python to measure how many bytes the integer `2` takes up, for example:

```
import sys

object = 2
sys.getsizeof(object)
```

For me, this outputs `28`, which means that our `object`—which is the integer `2`—takes up `28` bytes. (One byte is equivalent to eight bits.) I mentioned earlier that integers generally take up `32` bits, which should be only `4` bytes. However, I also mentioned that Python crams extra stuff into objects, so it's hard to always predict precisely how much space a particular object will take up.

To benchmark the space consumption of the various counting sort algorithms, I inserted the `sys.getsizeof()` method into the three different implementations of counting sort. Specifically, I called the `getsizeof()` function on the underlying data structure within each implementation. Here's what this looks like for the bit vector implementation:

```
import sys
import bit_vector


def counting_sort(array):
    set = bit_vector.BitVector(1000)

    for value in array:
        set.set_bit(value)

    print(sys.getsizeof(set.integers))
    return set.values()
```

Note that here I checked the space of `set.integers` rather than simply `set`. This is because in this context, `set` is a `BitVector` instance, and when we run `sys.getsizeof()` on a class instance, Python measures the space of the instance itself *without* all the data it references. To get the actual data, I had to run `sys.getsizeof(set.integers)`.

I then shuffled the numbers `0` through `999` and ran counting sort for each implementation (hash table, Boolean array, and bit vector). When I compare the space consumption of each implementation, I get:

Hash Table	36,960 bytes
Boolean Array	8072 bytes
Bit Vector	312 bytes

I think these results speak for themselves and show how incredibly tiny a bit vector can be, and yet still contain all of our set data.

Bit Vector Overkill

A bit vector isn't going to save you space in *all* scenarios. In particular, bit vectors are not ideal for *sparse sets*. A sparse set is a set in which there's a relatively large range of possible values, but only relatively few values across that range. For example, suppose the range of possible values is `0` through `999`. No matter how many values the set will ultimately hold, our bit vector will contain 1,000 bits to accommodate all 1,000 possible set values. So, even if our set only contained five values, our bit vector would take as much space as it takes to hold 1,000 values.

However, with a hash table, if we're only going to store five values, the hash table will be quite small. When I benchmark it, Python tells me that a hash table with five values takes up only 232 bytes of space, which is smaller than the 1,000-value bit vector that takes up 312 bytes.

It emerges that a bit vector is ideal for sets that are more *dense*, meaning that the number of values in the set is somewhat closer to the range of possible values. For example, a bit vector of 1,000 bits that holds 800 values is relatively dense. Specifically, it has a density of 80 percent.

You may have to do some benchmarking of the space within your particular application to figure out at what point it will be worth it to use a hash table vs. a bit vector.

No matter what, though, a bit vector will always be smaller than a Boolean array. This is because both data structures create as many pieces of data as there are in the possible range of values. However, a Boolean array will create a new Boolean object for each possible value, while a bit vector will create a new integer for every *32* values.

The Space Complexity of Sets

In discussing time or space complexity, we typically use the variable N to represent how many pieces of data we're dealing with. In the context of sets, N would represent the number of elements our set contains.

A hash table set has a space complexity of O(N) since the hash table's space is (constants aside) tailored to hold the set's N elements in N cells.

However, with Boolean arrays and bit vectors, the number of values they represent corresponds not to the number of *values* in our set, but rather the *range* of possible set values. So, even if a set has five values, if those values fall in a range from `0` to `1000`, our Boolean array has to hold 1,000 integers. To signify this, I'm going to use the variable R to represent the number of possible values in the range.

So, a Boolean array has a space complexity of O(R), as it holds up to R Boolean values of `True` or `False`.

A bit vector, on the other hand, can fit 32 values inside each of its integers on a 32-bit machine. Because this number can vary from machine to machine, we'll use the variable *B* to signify the number of *bits* that each integer holds.

It turns out that a bit vector stores a total of R/B integers. For example, if the range is 256, and the number of bits in an integer is 32, a bit vector needs only to contain 8 integers. This is because `256 / 32 = 8`.

Accordingly, we'd say that the space complexity of a bit vector is O(R/B). Some literature out there may use different variable names instead of R or B, but you get the idea.

Classic Set Operations

Because we're talking about sets, it's worth highlighting some of the most common set *operations*. These are the operations known as *union*, *intersection*, and *difference*. To keep things simple, I'll demo these operations using single integers as bit vectors, rather than using an array of integers.

Union

The *union* of two sets is a third set that contains all the values from the first two sets *combined*. For example:

```
Set A: {0, 3, 7}
Set B: {2, 3, 6}

Union: {0, 2, 3, 6, 7}
```

Note that the value `3` only appears once in the union. Although both Set A and Set B contain a `3`, the union set contains only one instance of `3`. This again is because a set, by definition, does not hold duplicate values.

Thanks to bitwise operators, it's easy and fast to find the union of bit vector sets. All we have to do is OR the two bit vectors together:

```
Set A:  10001001
Set B:  01001100

A | B = 11001101
```

That is, the OR operation will produce a `1` bit at any bit index where *either* bit vector has a `1` bit. This is the union of the two bit vectors.

Intersection

The *intersection* of two sets is a third set that contains only the values that the first two sets *have in common*:

```
Set A: {0, 3, 7}
Set B: {2, 3, 6}

Intersection: {3}
```

We can find the intersection of two bit vectors with a simple AND operation:

```
Set A:  10001001
Set B:  01001100

A & B = 00001000
```

The AND operation informs us where both bit vectors *share* a `1` bit. It is only the values represented by these `1` bits that get included in the intersection set.

Let's look at one more operation.

Difference

The *difference* between two sets is a third set that contains all items from the first set *minus* intersecting items from the second set:

```
Set A:                  10001001 -> {0, 3, 7}
Set B:                  01001100 -> {2, 3, 6}

Difference of A minus B: 10000001 -> {0, 7}
```

Here, the difference is identical to Set A, except that we removed the `3` since Set B also contains a `3`.

Interestingly, we can compute the difference between Set A and Set B by ANDing A with the *inverse* of Set B. That is, we run the code: `A & ~B`. Here's what this looks like:

```
Set A:            10001001
Inverse of Set B: 10110011

A & ~B =          10000001
```

Here's why this works. Our goal can be broken down into three subgoals:

- Subgoal #1: Our result integer should have the same `0` bits that Set A does. (This is because the result set should not contain any values that Set A doesn't have.)
- Subgoal #2: If both Set A and Set B share a `1` bit at a particular digit place, the result should have a `0` bit at that place. (This is the subtraction of Set B from Set A.)
- Subgoal #3: At any place where Set A has a `1` bit, and Set B has a `0` bit, the result should have a `1` bit at that place. (These are the values of Set A that do not get subtracted out.)

Here's how we achieve these three subgoals with the trick of `A & ~B`.

Subgoal #1 is accomplished because when we AND a `0` bit from Set A with any other bit, the result will always be a `0` bit.

We take care of Subgoal #2 because by inverting Set B, we turn all of its `1` bits into `0` bits. This means that all of the values that Set B represents are now `0` bits. When we AND these `0` bits with the `1` bits of Set A, the result is a `0` bit. This subtracts out the Set B values from Set A.

Finally, we knock Subgoal #3 out of the park because by inverting Set B, we take all of the values it *doesn't* have and flip them into `1` bits. When we AND these `1` bits with Set A's `1` bits, the result will be a `1` bit.

And so, the pithy line `A & ~B` cleverly produces the difference between Set A and Set B. Just wow.

Wrapping Up

Speaking of wow, we've covered so much in this chapter! We looked at sets, Boolean arrays, and bit vectors. We also looked at binary numbers, bitwise operators, and various bit manipulation techniques. We discovered how Boolean arrays can produce unique speed boosts, and how bit vectors can offer astounding space savings.

Although we implemented bit vectors from scratch, it's worth noting that there's also a popular Python bit vector library out there called bitarray.[7] You should definitely check that out if you're planning on using bit vectors in production.

There does appear to be a significant limitation to Boolean arrays and bit vectors. That is, they can only be used in scenarios where the possible set values are integers and these integers lie in a relatively small range. If, for example, our set values are *strings*, these data structures won't be very useful. Or so it would seem.

In the next chapter, we'll look at how we can harness the power of bit vectors *for all types of values*, strings included. With a little help from Monte Carlo algorithms, we can create a bit-vector-like data structure that works in many other scenarios beyond a small range of integers. Prepare to be amazed.

Exercises

The following exercises provide you with the opportunity to practice with bit vectors and bit manipulation. The solutions to these exercises are found in the section *Chapter 12*.

1. Write a function that accepts a binary string and returns the decimal number that the string represents. For example, if the function receives the string `"100101"`, the function should return the integer `37`. (Note that this is essentially the functionality of the Python expression `0b`. However, this exercise is to build the same functionality from scratch.)

2. This exercise is the inverse of the previous exercise. Here, you are to write a function that receives a decimal number and returns a binary string. *This binary string should always contain 32 bits*, even if that means that there will be many `0` bits on the left-most side. So, if the function receives the integer `37`, the function should return `"00000000000000000000000000100101"`. (This is essentially the same functionality as Python's `bin` command, but again, you are to build this function from scratch.)

3. Toward the end of this chapter, I explained how the set operations `union`, `intersection`, and `difference` work. However, I only demonstrated how they operate on two single integers. Let's get our `BitVector` class to perform these operations as well.

   Specifically, add three new methods to our `BitVector` class. The `union` method should allow a `BitVector` instance to accept a second `BitVector` instance and return a third `BitVector` instance that represents the union of the first two bit vectors. The `intersection` and `difference` methods should work similarly, except that they return a bit vector representing the intersection and difference, respectively. To keep things simple,

you can assume that the two bit vectors we're operating on are of the same size.

4. Write a function that accepts two integers and returns their *hamming distance*. The hamming distance of two integers is the number of digit places in which the two integers have different bits. For example, take the integers `1` and `8`. Their hamming distance is `2`, and here's why:

   ```
   1: 0001
   8: 1000
      ^  ^
   ```

   You can see that in two different digit places (the right-most and left-most places), the `1` and the `8` have different bits. Because the bits are different in *two* places, we say that `1` and `8` have a hamming distance of `2`.

   The integers `7` and `8` have a hamming distance of `4` since they have different bits in *four* digit places:

   ```
   7: 0111
   8: 1000
      ^^^^
   ```

   The key is figuring out what bitwise operations to utilize within your function.

5. *Puzzle:* This is one of my all-time favorite computer science problems. You are given an *unsorted* array that contains integers. It's guaranteed that in this array, each integer appears exactly *twice*—except for one integer that appears only *once*. For example, the array might be `[5, 9, 9, 3, 5]`. The `3` only appears once, but the `5` and `9` both appear twice.

   Write a function that returns the integer that only appears once in the array. For this example, your function should return the integer `3`.

This may sound simple enough, but there's a catch. Your function should run in O(N) time and take up no extra space. This rules out using a hash table or even a bit vector since those both take up space. It also rules out sorting since that takes O(N log N) time.

Once again, bitwise operators will be your friend. In fact, you may need one bitwise operator. Hint hint.

Footnotes

[6] https://pragprog.com/titles/jwpython2

[7] https://pypi.org/project/bitarray/

Chapter 13

Cultivating Efficiency with Bloom Filters

In the previous chapter, you discovered that bit vectors can store a lot of data in an incredibly compact way. In fact, one could argue that bit vectors are the most compact of all data structures. So, if you're looking to save space, bit vectors are an incredible tool.

However, you also learned that bit vectors are only useful in certain scenarios, specifically, where the data consists of integers that lie within a relatively short range. If you were to store integers in a large range or if you had other forms of data, such as strings, you wouldn't be able to take advantage of the great space savings bit vectors afford.

Fortunately, there's a variant of the bit vector that *can* handle even these other scenarios. In this chapter, you'll read about the Bloom filter, a data structure that acts like a bit vector but can save space for virtually *all* types of data. Bloom filters are incredibly clever and rely on several concepts we explored previously, including Monte Carlo algorithms and hash function families. By the time you're done with this chapter, you'll be able to implement your own basic Bloom filter and use it to save space across many different applications.

Finding Duplicates Revisited

Chapter 12, *Saving Space: Every Bit Helps* opened with the problem of how to find duplicates in an array of data. The problem was simple enough: write a function that, given an array, returns `True` if there are any duplicate elements, and `False` if there aren't. Using the brute-force approach, we grind along in $O(N^2)$ time, but we discovered that if we use a set to track what elements we encounter along the way, we can solve our problem at a brisk O(N) pace.

We also explored various options in selecting the right data structure for our set. Hash tables and bit vectors both gave us the O(N) result we were looking for, but bit vectors took up a lot less space than hash tables. However, we also learned that bit vectors can only store integers. This is because we use each bit's *index* to represent a value, and the index is itself an integer.

But let's say that our array contains strings, such as:

```
["apple", "banana", "cucumber", "date", "elderberry", "fig", "apple"]
```

These certainly aren't integers, so we can't store them in a bit vector.

Again, we *can* use a hash table, which works for any type of data, including strings. However, if we're strapped for space, and a hash table would simply be too large, is there anything we can do?

This would be an awfully short chapter if there weren't.

Bloom Filters

It turns out that even though a classic bit vector can't store strings compactly, a similar data structure called a *Bloom filter* can. The Bloom filter, named after its inventor, Burton Bloom, has been put to widespread use in all sorts of applications ever since the 1970s.

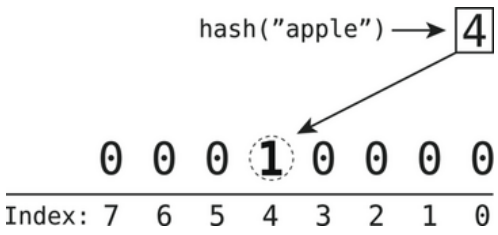
A Bloom filter is a space-efficient data structure used for storing a set. It's a variation of the bit vector, but can be used to store all sorts of data, not only a small range of integers. To make this work, Bloom filters make use of Monte Carlo principles, as you'll soon see.

To make it a tad easier to understand how Bloom filters work, I'm going to first introduce a similar but simpler data structure, which I call the *Gloom* filter. Please don't ever use a Gloom filter. I made it up for educational purposes only; it's not a real thing.

Gloom Filters

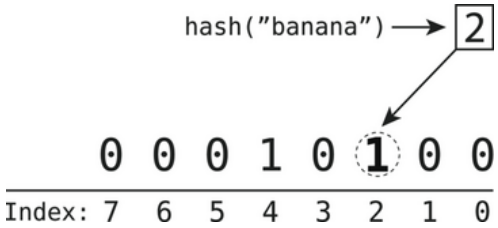
The Gloom filter utilizes *hashing* so that we can store strings inside bit vectors—or at least, kind of.

As you learned in Chapter 10, *Designing Great Hash Tables with Randomization*, we can use a hash function, such as division hashing, to transform a string into an integer. Suppose we choose a hash function that produces hash codes that are integers from `0` to `7`. Armed with such a hash function, we can store any string inside a byte-sized (8 bits) bit vector, as shown in the figure.



Because the hash code of `"apple"` is `4`, we flip the bit at index `4` to `1`.

Similarly, we can also add `"banana"` to our Gloom filter:



Our Gloom filter now contains both `"apple"` and `"banana"`.

And that, my friends, is the Gloom filter. In short, it uses a hash function to transform each value into an integer, and then sets the corresponding index in a bit vector to `1`. It's basically a bit vector, except that we first hash each value to decide which bit we should store it in.

Okay, that seems pretty cool. Let's analyze what we can and cannot accomplish with a Gloom filter.

One-Directional

If we had a Gloom filter and wanted to know what values it contains, we'd have absolutely no way to do that. The `1` bits in the byte `00010100` could represent *anything*. They could represent *any* string that hashes into `2` or `4`. A bit vector, on the other hand, only stores integers, and so `00010100` can only represent the integers `2` and `4`. As such, we can always draw out the integer values from the bit vector.

However, the fact that Gloom filters are one-directional doesn't mean that they can't be useful. Take the finding duplicates problem, for example. We never need to pull values out of the set; we only need the set to tell us if we've encountered a value before. So, if the Gloom filter returns `True` when we ask it if we've encountered `"apple"` before, we'll know that we've found a duplicate.

But there's a pretty big problem with this.

Collisions

As I covered in Volume 1, Chapter 8, and touched on in Chapter 10, *Designing Great Hash Tables with Randomization*, hashing can lead to collisions. That is, multiple values can all be hashed into the same hash code. This presented a potential problem for hash tables, but we were able to deal with collisions by using approaches such as separate chaining, which I discussed in Volume 1, Chapter 8.

However, collisions can derail Gloom filters since collisions cause *false positives*. To see what this means in our context, let's continue with the example of finding duplicates.

In the previous Gloom filter example, we encountered and stored the strings `"apple"` and `"banana"` in our Gloom filter, giving us the byte `00010100`. Cool.

Say that the next item in our array is `"cucumber"` and that `"cucumber"` hashes to `4`. When we check our Gloom filter to see whether we've encountered `"cucumber"` before, we'll find that there's a `1` bit at index `4`, and our code will tell us that we've already encountered `"cucumber"` before. In truth, though, this `1` bit was placed there because of `"apple"`, which happens to hash to the same value. At this point, our program will happily shut down and tell us that we found a duplicate—but we haven't!

This is what I mean by a false positive; the computer *thinks* it found a duplicate since it's confusing `"cucumber"` with `"apple"`, as they have the same hash codes. In any case, this problem of false positives is, well, problematic, as it means that a Gloom filter cannot help us detect duplicates properly.

No False Negatives

Now, I'd like to point out that even though Gloom filters suffer from false positives, it doesn't mean that Gloom filters are utterly useless. In particular, here's one thing that Gloom filters have going for them: they don't produce false *negatives*.

That is, if a Gloom filter declares that a value does *not* exist in the set, then the Gloom filter is 100 percent trustworthy. The reason for this is logical. A Gloom filter states that a value is nonexistent when the value's hash code corresponds to an index with a `0` bit. If the value was seen before, that index would definitely contain a `1` bit. If there's a `0` bit, we can be guaranteed that we've never encountered that value before.

To recap: a Gloom filter can produce false positives, meaning it may think that a value is a duplicate even though it's not. This can happen when two different values share the same hash code. However, a Gloom filter *cannot* produce false negatives. Accordingly, if the Gloom filter claims that the value does not exist, we can absolutely believe that.

Blacklists

Believe it or not, false positives can potentially be tolerable in a number of applications. One example of this is a *blacklist*.

Suppose we're maintaining a web server and the server has been suffering from a number of malicious attacks. Luckily, we've been able to check the logs and discover that these attacks are coming from a relatively small set of IP addresses. To combat the nefarious hackers, we create a blacklist of these IP addresses and tell the server to deny access to all of them.

To save space, we can store these IP addresses inside a Gloom filter. Then, each time our server receives a web request, we check the request source against our blacklist. That is, we hash the IP address into an integer and check the corresponding index inside our Gloom filter to see if there's a `1`

bit. If there is, it means that this IP has previously been blacklisted, and we will therefore deny access to the request.

Now, because there may be false positives, it's possible that we end up blocking a request from a benign IP address. This can happen if the benign IP address hashes into the same hash code as an address from our blacklist. However, depending on the application, *this may be okay*. That is, it may not be a big deal for us to block a few valid requests here or there in the name of defending ourselves from hackers.

At the same time, our blacklist is ironclad, as we will never end up accidentally allowing a request from an IP address on our blacklist. Because a Gloom filter reports no false negatives, if it says that the current request's IP address is not on the blacklist, we can trust it fully.

The Return of Monte Carlo

So far, we've seen that a Gloom filter can be used to represent a set. The Gloom filter is kind of like a bit vector, but it can be used to store even non-integer data (like strings)—something that bit vectors cannot do.

Sure, a hash table can also be used to store strings as a set. The advantage of Gloom filters, though, is that they take up less space than hash tables. That being said, Gloom filters have the *disadvantage* of potentially producing false positives.

You learned in Chapter 9, *Counting on Monte Carlo Algorithms* that a Monte Carlo algorithm can sacrifice accuracy in order to gain *speed*. It emerges that Gloom filters use another type of Monte Carlo algorithm. That is, they sacrifice accuracy to gain *space*. We could even say that a Gloom filter is a *Monte Carlo data structure* since it's a data structure that operates on Monte Carlo principles.

But is a Gloom filter a *good* Monte Carlo data structure? Indeed, all Monte Carlo algorithms and data structures, by definition, sacrifice some accuracy.

But to be an effective Monte Carlo algorithm, we need to ensure that the chances of error are low enough to satisfy our application's needs. To determine whether Gloom filters are effective, we need to measure how accurate or inaccurate they are.

So, let's do that.

Gloom Filter Error Rate

In the previous example, where all values hash into one of eight possible hash codes, any two values will have a 1 in 8 chance of colliding. This means that if we have a single item in our Gloom filter, when we check to see if some other item is in the Gloom filter, there's a 1 in 8 chance we'll get a false positive.

Now, say that there are four different items already in our Gloom filter, and they're all marked by different bits. The next item we check against our set will have a 4 in 8, or 50 percent, chance of being a false positive. That's a pretty high error rate. If you're using a Gloom filter for a blacklist, you'd be wiping out half of all legitimate traffic, which may be too much.

The question, then, is how we can reduce the false positive error rate of Gloom filters.

Increasing the Gloom Filter Size

Perhaps the easiest approach to reduce Gloom filter error rates is to simply increase the size of the Gloom filter. If we made it so that our Gloom filter uses 100 bits and the hash function produces 100 possible hash codes, there's only a 1 percent chance that two different values will collide. And if we make the Gloom filter the size of 10,000, the chances of any two values colliding will be a mere 1 in 10,000. Whether or not that's okay for your application is up to you, but this is most certainly a great improvement.

The downside, though, of increasing the Gloom filter's size is the fact that now we're taking up more space. The entire point of the Gloom filter was to reduce space, so by increasing its size, we're neutralizing the Gloom filter's main advantage. It's pretty hard to get a Gloom filter to an acceptable level of accuracy while still allowing us to save significant space.

But we can do better. In the next section, I'll introduce a new technique for reducing the filter's error rate. With this approach, we can turn our Gloom filter into a *Bloom* filter and achieve some impressive results for striking a solid balance between accuracy and space savings.

Use Multiple Hash Functions

We can turn our Gloom filter into a Bloom filter by using one deviously clever trick. And that is, we hash each value *more than once* upon each insertion and lookup. Let me explain.

You learned about hash function families in Chapter 10, *Designing Great Hash Tables with Randomization*. That is, we can create different hash functions that all use the same underlying hashing method and yet produce different hash codes. In that chapter, we created multiple hash functions that all use division hashing, but each hash function uses a different random prime number as part of its hashing calculation.

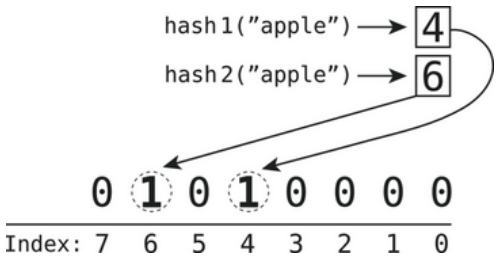
The main point I'm driving at here is that we can run a single value through a number of different hash functions, and each hash function will compute a different hash code. For example, let's say we have one hash function called `hash1()` and another function `hash2()`. If we use each one to hash the string `"apple"`, we may get something like this:

```
hash1("apple") ⟶ 4
hash2("apple") ⟶ 6
```

When a Bloom filter is first initialized, it decides how many hash functions it'll use, and what those hash functions are. (You'll see later how those decisions are made.) This is all decided at the Bloom filter's creation, and from then on, the Bloom filter uses these same hash functions for all of its operations, including lookups and insertions.

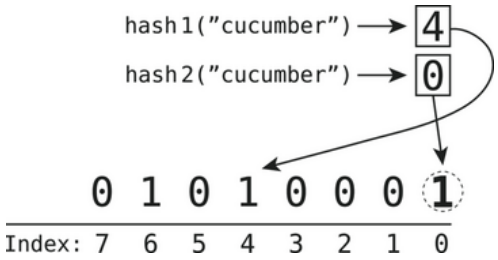
Let me demonstrate this with an example. Say that when we set up a Bloom filter, it decides that it will always use the two hash functions `hash1()` and `hash2()`.

If we want to insert `"apple"` into our Bloom filter, we'll run both hash functions and end up setting *two* bits to `1`:



Because `hash1("apple")` is `4`, and `hash2("apple")` is `6`, we set the bits of indexes `4` and `6` as `1` bits.

Now, let's say that we next insert `"cucumber"`. We might get something like this:



Here, our two hash functions compute the hash codes of `4` and `0`. So, the bit at index `0` gets set to `1`. We'd also set the bit at index `4` to `1`, but it already happens to be `1`. But that's okay—these types of collisions won't prove to be too much of a problem.

Insertions aren't the only operation where we use these two hash functions. When we look up an item in the Bloom filter, we also use both hash functions. That is, if we look up `"cucumber"`, we use the two hash functions to create two hash codes corresponding to two bit indexes.

Now, here's the key: only if *both* indexes have a `1` bit do we confirm that `"cucumber"` is present in our Bloom filter. If any of these bits were `0`, that proves that `"cucumber"` is *not* in our set. If it *were* in our set, both bits would have been set to `1`.

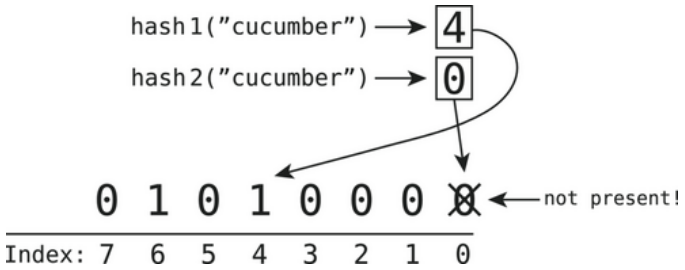
The Multi-Hash Advantage

Here's the advantage of using multiple hash functions. In the previous section, when we were only using a single hash function, we hit a false positive when trying to look up `"cucumber"`. This is because both `"apple"` and `"cucumber"` shared the same hash code of `4`. So, if `"apple"` already flipped the bit at index `4` to `1`, it appears that `"cucumber"` is in the set too, even though it isn't.

But now that we're using *two* hash functions, there's less of a chance of getting this false positive. Let's go back to the case where only `"apple"` is present in our set, and the Bloom filter looks like this:

```
    0 1 0 1 0 0 0 0
Index: 7 6 5 4 3 2 1 0
```

If we now look up `"cucumber"`, we're no longer going to get a false positive as shown in the figure.



Although the bit at index `4` is a `1` bit, the bit at index `0` is a `0` bit, so we know that `"cucumber"` is not present in the set. In other words, with a single hash function, two strings have a 1 in 8 chance of sharing the same hash code. With two hash functions, though, there's only a collision if both strings share a set of *two* hash codes. Because there's only a 1 in 8 chance of a single hash code being shared, there's a 1 in 64 chance that *two* hash codes will be shared (`1/8 * 1/8 = 1/64`). And so, we significantly reduce the chances of receiving a false positive.

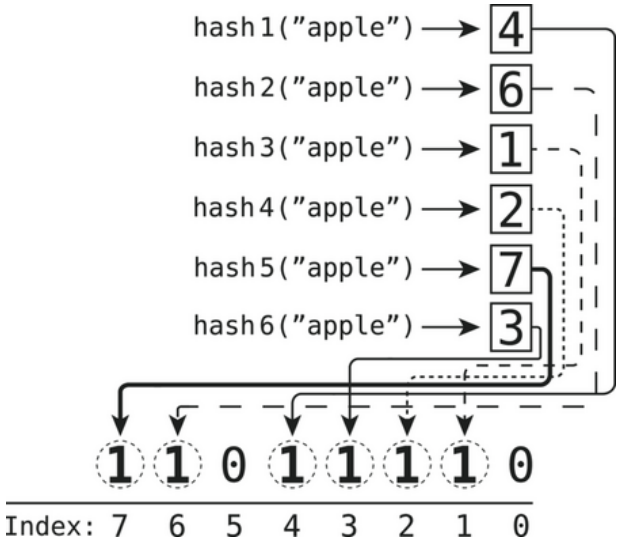
Too Much of a Good Thing

Because increasing the number of hash functions reduces the odds of getting false positives, it might be tempting to keep laying on the hash

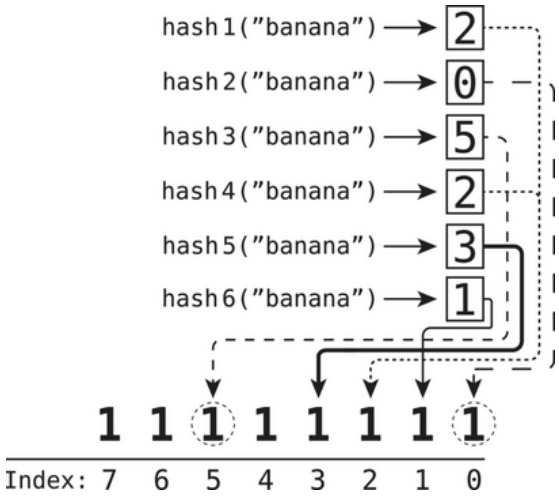
functions. Why stop at two hash functions when we can use six, eight, or ten? In theory, if we used 10 hash functions in our earlier example, the odds of two values sharing the same exact hash codes for *all 10 hash functions* should be 1 in 8 to the 10th power, which is 1 in 8,589,934,592.

However, a funny thing happens when we use too many hash functions.

Continuing with our earlier example, let's say that we used six hash functions for our byte of data. Here's an example of what would happen when we insert `"apple"`:

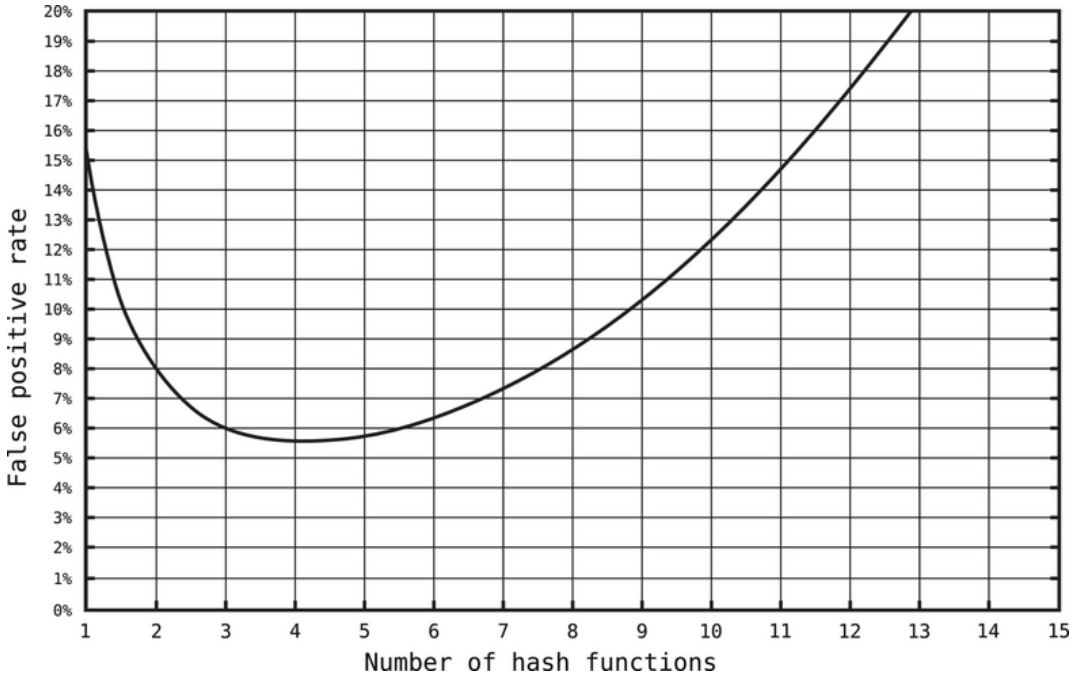


Almost the entire byte has been turned into `1` bits. And now here's what happens when we insert `"banana"` as shown in the figure.



Yikes—now the byte consists entirely of `1` bits! From this point on, *any string* we'd look up in the set would return `True` whether we ever inserted that value into the set or not. The only way to get a result of `False` is by encountering a `0` bit, but that could never happen anymore.

It's a curious thing, but it's true. Increasing the number of hash functions in a Bloom filter helps reduce false positives, but if we increase the number of hash functions by *too* much, we end up *increasing* false positives. The following graph illustrates what this looks like.



In this example, having just one hash function leads to a 15 percent false positive rate. As we increase the number of hash functions, the false positive rate drops. The false positive rate is at its lowest (about 5.5 percent) when we have four hash functions. However, as we continue to increase the number of hash functions beyond four, the false positive rate starts climbing higher again.

Bloom filters rely on a mathematical formula to calculate the ideal number of hash functions needed to keep both memory consumption and false positives to a minimum. So you, as the programmer, don't need to fret

about choosing the right number of hash functions. Once you program your Bloom filter correctly, it makes that decision for you.

Let's dig into how that works.

The Bloom Filter Variables: N, M, K, and F

Throughout the remainder of this chapter, I'm going to refer to four variables that factor into making sure our Bloom filter is optimized. You're already familiar with all the underlying concepts; I'm now assigning each concept to a variable:

- The variable N refers to the *number of items* we want our Bloom filter to hold.
- The variable M refers to *how much memory* our Bloom filter will take up, in terms of the number of bits.
- The variable K refers to *the number of hash functions* our Bloom filter will be using.
- The variable F refers to the *false positive rate*. If F is `0.01`, for example, this represents 1 percent. That is, for every 100 times our Bloom filter returns `True` as a result of a lookup, one of those results is expected to be a false positive.

Like or hate these variable names, these are what are used in the industry, so we'll run with them.

It turns out that these four variables are all connected to each other through a mathematical formula. It can be expressed in Python like so:

```
f = (1 - math.e**((-k * n) / m))**k
```

The variables `f`, `k`, `n`, and `m` refer to the F, K, N, and M variables we've been discussing. So, if we'd fill in `n`, `m`, and `k` with actual numbers into the code

and run it, the result will be the value for `f`, which, again, is the false positivity rate. (If you're wondering, `math.e` refers to a mathematical constant known as *Euler's number*, which is approximately `2.71828`.)

I'm not going to explain the mathematical theory behind this equation here. However, I *will* show you what this formula means in practical terms.

We discovered earlier that the more bits our Bloom filter contains, the lower F will be. That is, as we increase M, we have more hash codes available to us since each hash function will produce a hash code from 0 up until M. Accordingly, we reduce the chances that any two values end up with the same hash code.

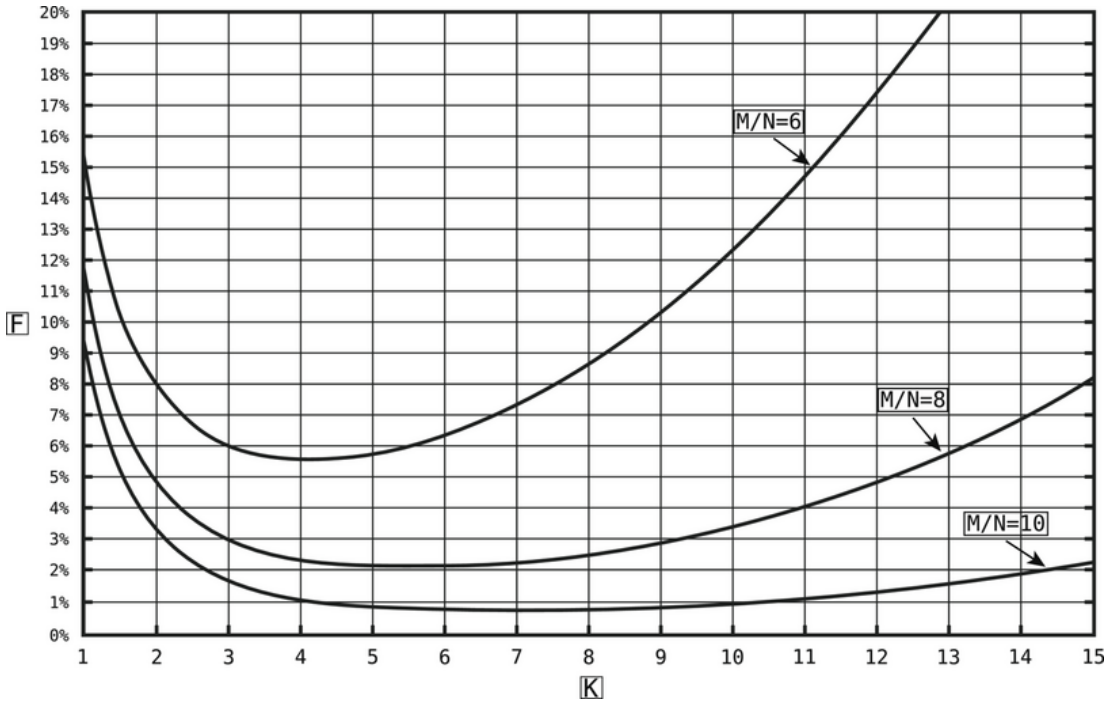
At the same time, increasing M only helps if N is significantly lower than it. Imagine that M was 1,000; that is, we have 1,000 bits inside our Bloom filter. If N is 10,000, we're still going to have way too many collisions since we're trying to cram 10,000 values into 1,000 slots. So, the thing to focus on is the *M/N ratio*—how many bits our Bloom filter will store relative to the number of items it'll hold.

For example, if we'll be dedicating 100 bits to our Bloom filter and only inserting 10 values, the M/N ratio is:

```
(M / N) = (100 / 10) = 10
```

And so, M/N being 10 means that we're going to have 10 times as many bits as there are values in our set.

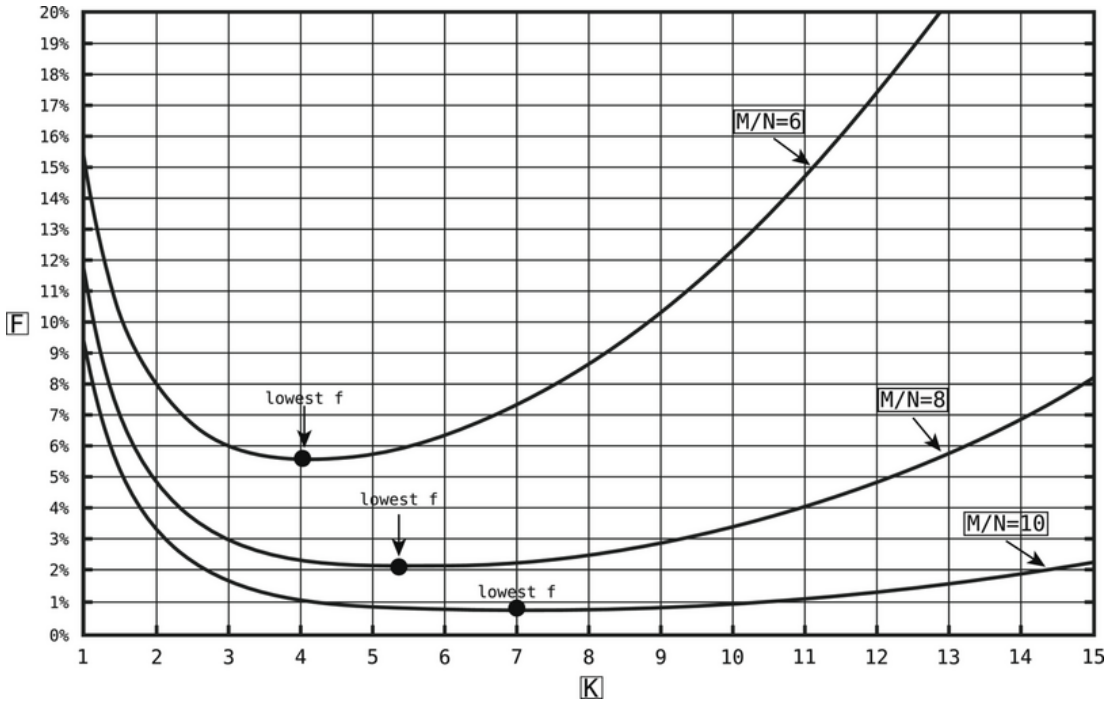
Now, the greater M/N is, the lower F is. For example, when M/N is 10, F will be lower than when M/N is, say, 6 or 8. The following graph illustrates this idea:



Here we can see how M, N, and K all affect F. When M/N = 6, meaning that there are six times as many bits as there are values in the Bloom filter, this produces one particular curve. However, you can see that the curve of M/N = 8 reaches lower levels of F. And M/N = 10 reaches yet even lower levels of F, yielding a false positivity rate that can be even lower than 1 percent.

You can also see from here that we can't determine what F is from M/N alone. M/N only produces the *curve* of what F can potentially be. It's *K* that finally pegs down what F is.

Take a look at the following graph. It's the same graph shown earlier, but here I highlight what K should ideally be for each curve (approximately):



Note that the ideal K changes based on what M/N is. When M/N is 6, it turns out that F dips to its lowest point when K is about 4. When M/N is 10, though, the ideal K is around 7.

Next up, we're going to see how to set up our Bloom filter so that it figures out for itself what its own M, F, and K should be.

The Classic Bloom Filter Constructor

The classic way to set up a Bloom filter is that upon creation, we pass in the variables N and F. This means that we need to decide in advance how many values (N) we plan on storing inside our Bloom filter. At the same time, we also need to decide the false positive rate (F) that we're willing to tolerate for our application.

For example, we may decide that we're going to store 100 items and that we're only willing to tolerate a false positive rate of up to 3 percent. Our Bloom filter's constructor starts like this:

```
class BloomFilter:
    def __init__(self, n, f):
```

```
        # remaining code will be filled in soon
```

And we'd initialize a new Bloom filter this way:

```
bf = BloomFilter(100, 0.03)
```

That is, N is `100`, and our maximum tolerable F is `0.03`.

While it may be tempting to input a super-small F such as `0.00000001`, this can only be achieved with a tremendous amount of memory (M). Given that we're using a Bloom filter to conserve memory, we want to balance our false positive rate with keeping the memory footprint small.

Code Implementation: Bloom Filter with the Classic Constructor

Without further ado, here is a basic Python implementation of a Bloom filter:

```
import bit_vector
import division_hasher
import math


class BloomFilter:
    def __init__(self, n, f):
        self.m = int(-math.log(f) * n / (math.log(2)**2))
        self.k = int(self.m * math.log(2) / n)

        self.hash_functions = []
        self.hash_function_primes = {}

        for _ in range(self.k):
            hasher = division_hasher.DivisionHasher(self.m)
            while hasher.prime in self.hash_function_primes:
                hasher = division_hasher.DivisionHasher(self.m)

            self.hash_function_primes[hasher.prime] = True
            self.hash_functions.append(hasher)
```

```
        self.bv = bit_vector.BitVector(self.m)

    def insert(self, value):
        for hash_function in self.hash_functions:
            hashcode = hash_function.hash(value)
            self.bv.set_bit(hashcode)

    def read(self, value):
        for hash_function in self.hash_functions:
            hashcode = hash_function.hash(value)
            if not self.bv.read_bit(hashcode):
                return False

        return True
```

This is the classic Bloom filter I described in the previous section, which accepts two parameters: `n` and `f`. That is, the programmer decides in advance approximately how many items the set will contain and what the maximum tolerable false positive rate should be.

With `n` and `f` in hand, the constructor first computes `m` with this dandy formula:

```
self.m = math.floor(-math.log(f) * n / (math.log(2)**2))
```

This formula is derived from the other formula we encountered earlier in *The Bloom Filter Variables: N, M, K, and F*. In any case, our Bloom filter now knows how much memory to allocate for our application.

Now that we've set both `n` and `m`, we also know our M/N ratio. Effectively, the Bloom filter has selected its desired curve from the graph.

Once the curve has been selected, all it needs to do now is compute `k` so that we hit the lowest possible `f` along that curve. The computation for `k` is:

```
self.k = math.floor(self.m * math.log(2) / n)
```

This formula, too, is derived from the original formula.

Now that we have `k` in hand, which, again, is the number of hash functions our Bloom filter will use, we now have to select the actual hash functions we'll be using. We store these hash functions in the array `self.hash_functions`.

To create the hash functions, we bring in the `DivisionHasher` class we created back in *Code Implementation: Randomized Hash Functions*. Our strategy is as follows: all of our hash functions will use division hashing. However, each hash function will incorporate a different random prime number into its division computation. (You can feel free to choose a different hashing scheme if you'd like, though. I'm simply using division hashing because it's simple, and we've covered it before.)

To make sure we don't accidentally create two identical hash functions, we keep track of each hash function's prime number in the hash table `self.hash_function_primes`. Then, the following code loops until it has created `k` different hash functions:

```
for _ in range(self.k):
        hasher = division_hasher.DivisionHasher(self.m)
        while hasher.prime in self.hash_function_primes:
                hasher = division_hasher.DivisionHasher(self.m)

        self.hash_function_primes[hasher.prime] = True
        self.hash_functions.append(hasher)
```

The inner `while` loop keeps generating a random hash function (in the variable `hasher`) until we get one that we haven't already created before. When we find an acceptable hash function, we append it to our array `self.hash_functions`.

Finally, our constructor creates a new bit vector and stores it in the variable `self.bv`. Here, we're using the bit vector class we created in *Code Implementation: Bit Vector*.

The rest of the class is relatively smooth sailing. The `insert` method iterates over each of the Bloom filter's hash functions and uses each one to hash the `value` into a hash code. For each hash code, we then flip the corresponding bit (based on its index) in the bit vector to `1`.

Similarly, the `read` method uses the same set of hash functions to hash the `value` and checks whether each hash code has a corresponding `1` bit inside the bit vector. If all the bits we check are `1`, we return `True` to indicate that the `value` is currently in our set. However, if even a single bit is `0`, we know that the `value` is *not* in the set and we therefore return `False`.

An Alternative Constructor

A potential downside with the "classic" Bloom filter constructor is that we don't get to tell the Bloom filter how much memory it should consume. Instead, we tell it what N and F are, and it computes M based on the mathematical formula. But what if we are absolutely constrained for space? It could happen that the Bloom filter may take up more memory than we're able to handle.

If you find yourself in this predicament, you may consider using an alternative constructor for the Bloom filter. Specifically, when we initialize the Bloom filter, instead of passing in the variables N and F, we pass N and *M*. By passing in M, we are dictating to the Bloom filter the absolute maximum amount of space that it should take up.

When we do this, though, we don't get to control F. That is, when the programmer chooses N and M, it is effectively choosing which curve on the graph the Bloom filter will use. The Bloom filter then computes the appropriate K to reduce F as much as possible. However, we're giving up our liberty to choose what F is.

For example, if we choose that M/N is 6, the best possible F we can achieve is 4 percent, period. Hopefully, that'll be okay for your application. If that's

not okay, a Bloom filter is not going to be a good fit for your software.

Ultimately, we have two choices. With the classic constructor, we choose what F we're willing to tolerate, and the Bloom filter chooses the lowest possible M. With the alternative constructor, we choose what M is, and we have to deal with whatever F ends up being.

In theory, there could be yet other constructors, such as choosing F and M, and let the math choose N. However, the two constructors I've described so far are the most common, so we'll stick with those.

Code Implementation: Bloom Filter with an Alternative Constructor

Here is what the alternative constructor looks like:

```
class BloomFilter:
    def __init__(self, n, m):
        self.m = m
        self.k = int(m * math.log(2) / n)
        self.f = (1 - math.e**((-self.k * n) / m))**self.k

        self.hash_functions = []
        self.hash_function_primes = {}

        for _ in range(self.k):
            hasher = division_hasher.DivisionHasher(self.m)
            while hasher.prime in self.hash_function_primes:
                hasher = division_hasher.DivisionHasher(self.m)

            self.hash_function_primes[hasher.prime] = True
            self.hash_functions.append(hasher)

        self.bv = bit_vector.BitVector(self.m)
```

This constructor has the arguments of `n` and `m`. The variable `k` is computed using the same code as the classic constructor, as only `n` and `m` are needed to compute `k`.

The code then computes F. In truth, F isn't used anywhere else in the code. However, it's useful to have so that a programmer can check what F ends up being and can decide if it's tolerable.

Bloom Filter Deletion

One important thing to note about Bloom filters is that they don't have a deletion operation. This is because deletion would be impossible for the following reason:

In theory, we'd delete a value by hashing the value into multiple hash codes and flipping all the corresponding 1 bits to 0. The problem, though, is that other values in the set likely share some of these 1 bits. So, by deleting one value, we'd be inadvertently deleting other values as well.

Using Bloom Filters for Detecting Duplicates

It's time to turn our attention back to the problem we discussed at the beginning of this chapter. We have an array of strings and want to check whether there are any duplicates. Let's now solve this problem in a space-efficient way by using a Bloom filter.

The strategy is to iterate over the array of strings, and if a string is deemed not a duplicate, we insert it into a Bloom filter. To determine whether a string is a duplicate, we look it up in the Bloom filter to see if we've ever encountered it before.

Here's the code to implement this approach:

```
import bloom_filter


def find_duplicates(array):
    set = bloom_filter.BloomFilter(len(array), 0.01)

    for item in array:
        if set.read(item):
            return True
        else:
            set.insert(item)

    return False
```

If this code gives the result of `False`, we know with certainty that there aren't any duplicate values. On the other hand, if the code returns `True`, then it's likely a duplicate. As to how likely, well, that depends on the false positive rate we set for our Bloom filter. If F is 1 percent (as it is for this code example), then it's 99 percent likely that there's a duplicate.

And so, if your application is willing to tolerate errors 1 percent of the time, a Bloom filter is a great way to save space in storing your set while finding

duplicates.

Indeed, when I use the `sys.getsizeof()` method to measure the space of different sets, I find that to contain 100 values, a hash table takes up 4,688 bytes of space, while a Bloom filter takes up 296 bytes. This is true even though I've reduced the Bloom filter's false positive rate down to 1 percent.

Bloom Filters in the Wild

You've seen how Bloom filters work wonders in terms of saving memory. Because of this, Bloom filters are ubiquitous in the real world. In addition to the use case of a blacklist of web traffic sources we looked at earlier, the following sections highlight some further applications where Bloom filters are used.

Reducing Database Lookups

Many database engines use Bloom filters to help reduce lookup time. Looking up an item in the database usually involves reading information from the disk and, therefore, takes a significant amount of time relative to looking something up in main memory.

Now, this time lag takes place upon each lookup, no matter whether the item we're looking up is in the database or not. As such, it's kind of a waste of time to execute a database lookup to find an item that's not there. After all, there's no actual information we need to extract from the database in such a case. Wouldn't it be nice if we could know in advance that the item doesn't exist so we can avoid the lookup cost?

This is where Bloom filters can help tremendously. Each time an item is added to a database, that same item is *also* inserted into a Bloom filter that lives in main memory. We don't have to insert all the item's information into the Bloom filter; we can simply insert the item's name (or whatever its primary ID is).

The Bloom filter can then be used as a kind of index telling us what values can be found in the database. Each time we perform a lookup, we first check the Bloom filter to see if our desired item is in the database. Because the Bloom filter lives in main memory, this check happens extremely quickly. Only when the Bloom filter tells us that the item is present do we

then perform the more expensive operation of looking it up in the database. On the other hand, if the Bloom filter tells us that the item is *not* in the database, we get to skip the expensive database lookup.

As to the issue of the Bloom filter's possibility of error, there's not much downside, and here's why: If the Bloom filter tells us that the item doesn't exist, we can be absolutely sure that this is true since there can never be a false negative.

Now, there's a small chance for a false *positive*, meaning the Bloom filter will tell us that the item is present even though it isn't. However, the worst thing that could happen is that we perform an unnecessary database lookup. And this is not a big deal, especially considering that if we didn't use a Bloom filter at all, we'd be performing way more unnecessary database lookups! In other words, although a Bloom filter may not completely eliminate all unnecessary lookups, it will nevertheless still greatly reduce their frequency.

Again, a hash table can be used in this case instead of a Bloom filter, but the hash table would take up more space. And so, this is why many database engines use Bloom filters in practice to reduce database lookups.

Caches

In Chapter 4, *Cache Is King*, you learned all about caching. One of the primary lessons was that because caches only hold a limited amount of data, we only keep data that will be requested again in the future. On the flip side, we evict data that will never be asked for again.

The ultimate item we don't want to cache is something often called a "one-hit wonder." A one-hit wonder is a piece of data that is only being requested once in history; no one will ever request it again.

Now, there are some applications where most requests are indeed one-hit wonders. If, say, 75 percent of requests are for one-hit wonders, then we

only want to cache the other 25 percent of requests.

Of course, we can't know for sure whether a given request is a one-hit wonder, since we can't see into the future to know for certain that it'll never be requested again. However, we *can* use a Bloom filter to help prevent inserting one-hit wonders into our cache.

The strategy here is to only cache data from a request we know has been asked for *more than once*. That is, if this is the first time we ever had such a request, we suspect that perhaps this is a one-hit wonder. However, if this is already at least the *second* time we encounter this request, then we know it's certainly not a one-hit wonder; after all, it's now been requested twice! And so, we'll cache the data, with the guesstimate that this request may be made yet again in the future.

Specifically, we'll use a Bloom filter to keep track of what requests have ever been made previously. Whenever a request is made, we insert the requested value into our Bloom filter. This will indicate to us in the future that this request has been made once so far. Then, going forward, each time a request is made, we first check the Bloom filter to see whether the request has been made before. If it was never made before, we suspect that this may be a one-hit wonder and do not cache the data. But if the Bloom filter tells us that this request *has* been made before, then we do cache the data.

Now, when the Bloom filter tells us that the request has never been made before, we can 100 percent believe this to be true. On the other hand, if the Bloom filter tells us that this request *has* been made before, there's a small chance that this isn't true.

However, the worst thing that can happen is that we cache a one-hit wonder. This is no travesty, though, especially given that this will happen only occasionally. And again, without the Bloom filter, we'd be caching *all* one-hit wonders! Although a hash table would be more accurate (that is, 100

percent accurate), it may take up too much space to be worth it. The Bloom filter is a much more compact data structure.

Variants

Besides the classic Bloom filter, there are plenty of other variants out there as well. These include Bloomier filters (yep, that's their name), Spatial Bloom filters, Scalable Bloom filters, Layered Bloom filters, and more. Each of these deals with specialized use cases, and you may enjoy researching them to see how the general Bloom filtering strategy can be used for so many different types of applications.

We've also seen that a Bloom filter is a Monte Carlo data structure. But it's certainly not the only one. While I don't have the space to cover other Monte Carlo data structures in this volume, I encourage you to check them out. Some of these include skip lists, quotient filters, cuckoo filters (totally!), count-min sketches, and HyperLogLogs.

Wrapping Up

Bloom filters can be used to save space even in instances where bit vectors cannot. A bit vector is generally limited to cases where our data is integers within a small range, but Bloom filters do not have these limitations. To keep themselves small, Bloom filters use the Monte Carlo approach—they sacrifice some accuracy in order to save space.

You've seen in previous chapters how Monte Carlo algorithms can also be used to gain speed. In short, then, the idea of Monte Carlo is to sacrifice some accuracy to gain either time or space.

In some cases, you may save both time *and* space. One example of this is using a Bloom filter to reduce database lookups (as described in *Reducing Database Lookups*). The main purpose of maintaining a set in main memory in this context is to save time. And by specifically using a Bloom filter to serve as the set, you're ensuring that your set takes up minimal space.

Parting Thoughts

Congratulations—you've leveled up! It's been quite a journey, and you've learned many ideas that you can take to the bank.

Understanding the limitations of Big O Notation and how to benchmark time and space are critical to ensuring that your algorithms are truly efficient.

Randomization and Monte Carlo play an important role in thoughtful algorithm design. External-memory algorithms and caching are both important components of good system design.

Data structures such as red-black trees and randomized treaps make sure that when you need a tree, it's fast. Bit vectors and Bloom filters, on the other hand, offer tremendous space savings.

Sure, I can spell out a list of other things you've learned, such as bit manipulation, the sliding-window technique, substring search, Mergesort, B-trees, and random hashing. However, above all, you've upgraded the way you *think*. You're a more adept software engineer, able to thoughtfully consider the various implications of each code implementation. You understand the pros and cons of each data structure and the advantages and disadvantages of various algorithms that all vie to solve the same problem.

And yet, your journey is still not over. There are many algorithms and data structures yet to learn, and efficiency-producing techniques and tricks that you have not yet encountered. As our computing problems get ever more complex, new solutions are constantly being developed, and others are yet to be discovered.

Your growth as a software engineer must continue! Whether it's through my own future writing or through your own research, I wish you the best of

luck as you progress along your journey.

Exercises

The following exercises provide you with the opportunity to practice with Bloom filters. The solutions to these exercises are found in the section *Chapter 13*.

1. You're the lead engineer at the social media app HumbleBrag. This particular app requires users, upon sign-up, to choose a unique username. If a proposed username is already found to be taken, the user is asked to choose a different username.

   Now, you've gotten complaints that this process is too slow. Each time a user suggests a username, it takes a whopping *four seconds* (sheesh!) for the app to tell the user whether that username is available. You know that this is happening because the app, under the hood, is calling the database to search whether that username exists.

   How might you use a Bloom filter to help speed things up?

2. Thanks to your ingenuity in using the Bloom filter, you made HumbleBrag's username validator much faster. Buoyed by this success, the rest of your engineering team got super excited to apply this technique to speed up other parts of the app.

   Another slow part of the HumbleBrag app is user login. Specifically, each time a user enters their password, the app needs to query the database to see if (an encrypted version of) that password is correct. The engineering team is suggesting storing encrypted passwords in a Bloom filter. This way, we could check the password using the super-quick Bloom filter rather than run a slow database lookup.

   Is this a good idea?

3. Use Python to create an instance of the “classic” `BloomFilter` class. Set it up to accommodate a set containing 1,000 values and to have a failure rate of 3 percent. Write code to help you answer the following questions:

   What does M come out to be?

   What does K come out to be?

   How many *bytes* does the underlying bit vector data take up when you call `sys.getsizeof()` on the bit vector’s underlying array?

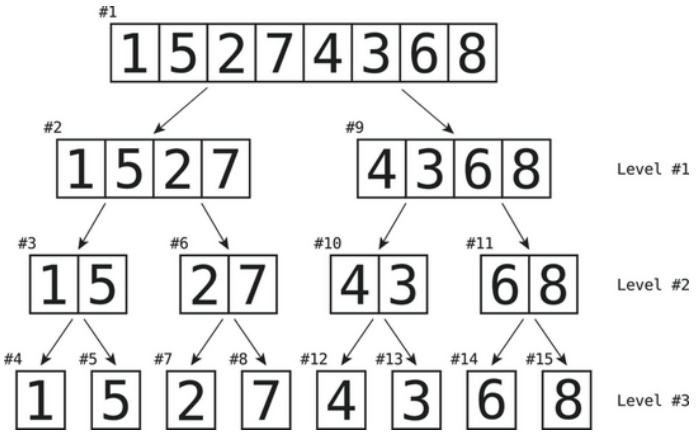
Appendix 1

Solutions

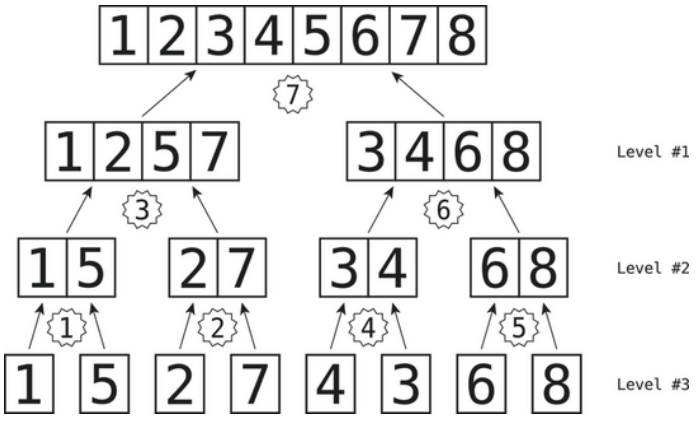
Chapter 1

These are the solutions to the *Exercises*.

1. Here's what the breakdown phase of Mergesort looks like for this example:



2. Here's what the merging phase of Mergesort looks like for this example:



3. Here's what the result of Mergesort looks like for this example:

The interesting thing to note here is that for each integer, the timestamps remain sorted! For example, although we moved around the three instances of value `1` so that they're now all in a row, their timestamps remain in their original order.

Because of this, computer scientists refer to Mergesort as an example of a *stable sort* algorithm. A stable sort algorithm ensures that when there are duplicate instances of the same value, those two instances will remain in the same order (relative to each other) that they were before the sorting.

Note that many sorting algorithms, including Quicksort, do *not* produce a stable sort. If stable sorting is important for your application, this may be reason enough to choose Mergesort over Quicksort.

4. As I mentioned in the exercise, there's no right or wrong way of doing this, but here's what I did. Here are my two loops, starting with the first version:

```
for i in range(1, 11):
    print(i)
```

And here's my second version:

```
x = 1
while x < 11:
    print(x)
```

Here are the two sets of bytecode:

```
For loop:

  1           0 LOAD_NAME                0 (range)
              2 LOAD_CONST               0 (1)
              4 LOAD_CONST               1 (11)
```

```
              6 CALL_FUNCTION            2
              8 GET_ITER
        >>   10 FOR_ITER                12 (to 24)
             12 STORE_NAME               1 (i)

  2          14 LOAD_NAME                2 (print)
             16 LOAD_NAME                1 (i)
             18 CALL_FUNCTION            1
             20 POP_TOP
             22 JUMP_ABSOLUTE           10
        >>   24 LOAD_CONST               2 (None)

While loop:

  1           0 LOAD_CONST               0 (1)
              2 STORE_NAME               0 (x)

  2     >>    4 LOAD_NAME                0 (x)
              6 LOAD_CONST               1 (11)
              8 COMPARE_OP               0 (<)
             10 POP_JUMP_IF_FALSE       22

  3          12 LOAD_NAME                1 (print)
             14 LOAD_NAME                0 (x)
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 JUMP_ABSOLUTE            4
        >>   22 LOAD_CONST               2 (None)
             24 RETURN_VALUE
```

Both sets of bytecode seem to have the same number of instructions, and similar ones at that. However, they clearly aren't exactly the same.

For example, the `for` loop version has the `FOR_ITER` instruction, which appears to be specific to a `for` loop. The `while` loop version has unique instructions such as `COMPARE_OP`, which is needed to compare `x` to `11`. This version also has a `POP_JUMP_IF_FALSE` command, which appears to terminate the loop if the expression `x < 11` is false.

Chapter 2

These are the solutions to the *Exercises*.

1. Personally, I would have guessed that the first version is faster. This is because the first version relies on linear search, which is O(N), while the second version relies on sorting, which is O(N log N). Indeed, my benchmarking bears out this hypothesis. Here is my benchmarking code:

```
import timeit

setup_code = '''
import random
import exercise_1a

array = []
for i in range(1000000):
    n = random.randint(1, 1000000)
    array.append(n)
'''

test_code = '''
exercise_1a.minimum(array)
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5,
number=1))
```

   This code benchmarks the first version, which I saved in a file called `exercise_1a.py`.

   When I benchmarked the first version, my results were:

```
0.029821541000000007, 0.030344957999999922, 0.03037758299999993,
0.029123583000000064, 0.030518541999999815
```

I saved my second version in a file named `exercise_1b.py` and updated my benchmarking code accordingly. The benchmarking results of my second version, as I'd predicted, were slower:

```
0.22374379099999997, 0.21845316700000006, 0.21908141599999986,
0.22122462500000006, 0.218094958
```

2. For this scenario, I predicted that the two functions would have the same speed. I based this hypothesis on the fact that in the exercises for Chapter 1, we saw that the bytecode for a `for..range` loop and a `while` loop were similar.

   But it turns out that my prediction was not entirely correct. Here is my benchmarking code:

```
import timeit

setup_code = '''
import random
import exercise_2a
'''

test_code = '''
exercise_2a.sum_up_to_one_million()
'''

print(timeit.repeat(stmt=test_code, setup=setup_code, repeat=5,
number=1))
```

   This code works just as well for the second version; I only needed to change the module name.

   Here are my benchmarking results for the `for..range` code:

```
0.05982770799999999, 0.05473741700000001, 0.054945541000000014,
0.05513195900000001, 0.055346958
```

   My benchmarking results for the `while` loop code showed that this second version was slightly slower:

```
0.087291291, 0.08183283300000001, 0.08237899999999998,
0.08279591600000002, 0.08296995899999998
```

As to why this is, well, that's hard to know for sure without delving into Python's source code to see how its loops are implemented. There's discussion on the Internet about this, and I invite you to do further research if you're interested in learning more.

3. The most important case to test when it comes to sorting is a list of jumbled values, which is the average case. In my setup code, though, I created an array of *sorted* integers. Sorting presorted values can either be much faster or slower than the average case, and it may be worthwhile benchmarking as well. However, it's certainly a mistake to *only* benchmark this edge case and ignore the performance of the typical case, which is to sort unsorted values. To fix this, the setup code should instead create an array of integers that are in *random* order.

4. First, I shouldn't be including the `print` command in my `testcode`, as printing takes time, and we're not interested in testing how fast printing is.

   Second, in my setup code for linear search, my array only contains 100,000 values, while my binary search benchmark creates an array of *one million* values. This isn't an apples-to-apples comparison, and is therefore an experiment that is not properly controlled. It's pretty messed up, to be honest.

Chapter 3

These are the solutions to the *Exercises*.

1. Here's one possible approach, which randomly selects 3 indexes from the array and returns the values at those indexes.

```
import random


def pick_3(array):
    chosen_indexes = []

    for _ in range(3):
        random_index = random.randint(0, len(array) - 1)
        while random_index in chosen_indexes:
            random_index = random.randint(0, len(array) - 1)

        chosen_indexes.append(random_index)

    # sort the indexes to ensure we return the values
    # in their original order:
    chosen_indexes.sort()

    chosen_values = []
    for index in chosen_indexes:
        chosen_values.append(array[index])

    return chosen_values
```

2. Here, we grab all the hash table's `keys()`, convert the result into an array with the `list` keyword, and then use `random.choice` to pick one random item from the array:

```
import random


def sample(hash_table):
    return random.choice(list(hash_table.keys()))
```

3. Let's say that we're selecting a single random value from the array `["a", "b", "c", "d", "e", "f", "g"]`. Our first iteration of the loop will point to the `"a"` and decide whether it's the value to select. The question is: how can we ensure that this `"a"` has an equal chance of being selected as each other value in the array?

   Now, because there are 7 values, this means that each value should have a 1/7 chance in being selected. And so, to decide if we're going to select the `"a"`, we should roll a 7-sided die, and if the die lands on one particular side, we'll select the `"a"`. If the die lands on any of the other 6 sides, we will *not* select the `"a"`.

   In code, this would look something like `roll = random.randint(1, 7)`, which chooses a random integer from `1` to `7`, inclusive. We can decide that if the result is a `1`, we'll select the `"a"`, and if the result is some other number, we'll move on to the next iteration of the loop.

   Now, in the next iteration of the loop, the current value is `"b"`. How do we give `"b"` an equal opportunity of being the chosen value? Like every other value, we want to ensure that the `"b"` also has a 1/7 chance of being selected.

   Intuitively, we may think that we should roll another 7-sided die to see whether we'll choose the `"b"`. However, this is not the right move to make, and here's why.

   The `"a"` already got its day in the sun—that is, we already gave it a fair chance at being selected. Once the `"a"` was *not* selected, it no longer has any bearing on whether we should choose the `"b"`. And so, when we are deciding whether to select `"b"`, we need to give the `"b"` the same odds as every other value *that we haven't iterated over yet*, namely, `"c"`, `"d"`, `"e"`, `"f"`, and `"g"`.

In other words, with `"a"` out of the way, we're now deciding whether, of the 6 values that remain, we should select the `"b"`. Therefore, we want to give the `"b"` a 1/6 chance of being chosen. And so, we can run `roll = random.randint(1, 6)` and select the `"b"` if the `roll` turns out to be `1`.

To make this even more intuitive, I'll explain this from yet another angle. Imagine that we're randomly selecting a value from an array that only contains 2 values, such as `["a", "b"]`. Say that we gave the `"a"` a 1/2 chance of being chosen, and it wasn't selected. At this point, we should simply select the `"b"` instead. The `"a"` lost its 50 percent chance, so we select the `"b"` instead.

If we were to instead roll the die again to see whether we select the `"b"`, the `"b"` will ultimately only have a 1/4 chance of being selected since 1/2 * 1/2 = 1/4. Additionally, if the `"b"` loses on its roll, we won't end up selecting anything from the array!

Similarly, with the case of `["a", "b", "c", "d", "e", "f", "g"]`, if we rolled a 7-sided die for `"a"` and do the same again for `"b"`, it would come out that the `"b"` has a 6/49 chance of being selected. That is, for the `"b"` to be selected, it relies on the 6/7 chance that the `"a"` will *not* be selected. When we multiply that 6/7 chance by the 1/7 that `"b"` *will* be selected, that produces a chance of 6/49. The odds of 6/49 are slightly less than the 1/7 we were aiming for.

So instead, we roll a *6*-sided die for the `"b"`. This way, when we multiply the 6/7 chance that the `"a"` is *not* selected by the 1/6 chance that the `"b"` *is* selected, we get: 6/7 * 1/6 = 6/42, which is the same as 1/7.

As we proceed through each value in the array, we reduce the "sides of the die" by 1. So on the next round, we'll roll a 5-sided die, and on the next round a 4-sided die, and so on.

If the first 6 values are not selected, we automatically select the final value, the `"g"`. The `"g"` relies on the 6/7 chance that the `"a"` isn't selected, multiplied by the 5/6 chance that the `"b"` isn't selected, multiplied by the 4/5 chance that the `"c"` isn't selected, and so on.

This gives us:

6/7 * 5/6 * 4/5 * 3/4 * 2/3 * 1/2 = 1/7. Tada!

Here is the code:

```
import random


def sample(array):
    denominator = len(array)

    for value in array[:-1]:
        roll = random.randint(1, denominator)

        if roll == 1:
            return value

        denominator -= 1

    return array[-1]
```

4. This solution builds upon the previous one. The gist of the algorithm goes like this:

    1. We begin to traverse the tree, starting at the root node. We'll use the variable `current_node` to refer to whichever node we're pointing to at a given moment.

    2. We always keep track of how many nodes are contained in the subtree of which the `current_node` is the "root." We'll call this variable the `subtree_size`. (At the beginning of our algorithm, when

`current_node` is the true root, the `subtree_size` will indeed be the size of the entire tree.) Since we know that the tree is complete, we can compute the `subtree_size` based on how many levels the tree has rather than traverse the entire tree and count all the nodes. See the `tree_size` method to follow for the exact calculation.

3. We roll a die from `1` up to the `subtree_size`. If the die lands on `1`, we select the `current_node` as our winning node. If, for example, the subtree contains 15 nodes, this gives the current node a 1/15 chance of being chosen. This is exactly what we want since we want to give each of the 15 nodes of the tree an equal chance of being chosen as the winner.

4. If our die does *not* land on `1`, then we continue to traverse the tree by moving down to the `current_node`'s child. To decide whether we'll select the left child or the right child, we roll a die. If it lands on `1`, we move left, and if it lands on `2`, we move right. This way, both the left descendants and the right descendants of the `current_node` have an equal chance of eventually becoming the winner. Whichever child we end up choosing becomes the new `current_node`.

5. We now calculate how many nodes are contained in the subtree of the new `current_node`. Since each time we move down a level in a binary tree, we exclude half of the remaining nodes from our traversal path, we simply divide the old `subtree_size` by `2`. (We use floor division, so if the previous subtree had 15 nodes, the current subtree now has 7 nodes. This floor division works since we also have to exclude the previous `current_node` itself, in addition to the other half of its descendants.)

6. We start over again at Step "c." That is, we now roll a die from `1` up to the new `subtree_size`. If we roll a `1`, the new `current_node` is

the winner; otherwise, we move on again. If we eventually reach a leaf node, the `subtree_size` will be `1`, in which case the leaf node will definitely be selected as the winner.

Here's the code, including a simple implementation of a BST:

```
import random


class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left_child = left
        self.right_child = right


def insert(value, node):
    if value < node.value:

        if not node.left_child:
            node.left_child = TreeNode(value)
        else:
            insert(value, node.left_child)

    elif value > node.value:

        if not node.right_child:
            node.right_child = TreeNode(value)
        else:
            insert(value, node.right_child)


def tree_size(node):
    level_size = 1
    total_size = 1

    current_node = node.left_child
    while current_node:
        level_size *= 2
        total_size += level_size
        current_node = current_node.left_child
```

```
        return total_size


    def sample(node):
        subtree_size = tree_size(node)
        current_node = node

        while current_node:
            roll = random.randint(1, subtree_size)
            if roll == 1:
                return current_node.value

            subtree_size //= 2

            roll = random.randint(1, 2)
            if roll == 1:
                current_node = current_node.left_child
            else:
                current_node = current_node.right_child
```

The primary algorithm here takes place in the `sample` method. You'll see, though, that it relies on a `tree_size` method to calculate the size of the entire tree. Again, this calculation is only valid if the tree is complete. Otherwise, you may have to traverse the entire tree and simply count up all the nodes.

Chapter 4

These are the solutions to the *Exercises*.

1. In the first five steps, we fill up our cache with 5 values:

   ```
   ["c", "t", "h", "o", "p"]
   ```

   Technically, these are all cache misses, so we have 5 cache misses so far. When we then request the next `"t"` and `"h"`, these are both cache hits since those values are currently in the cache.

   Next, we reach the first `"z"`, which brings our cache miss tally to 6. This will evict the `"c"` since the `"c"` will never be requested again. Our cache currently looks like this:

   ```
   ["z", "t", "h", "o", "p"]
   ```

   The next request is the second `"o"`, which is a cache hit. After that, we reach the `"a"`, which is our 7th cache miss. This will evict the `"o"`, which won't ever be requested again.

   Our cache now looks like this:

   ```
   ["z", "t", "h", "a", "p"]
   ```

   The next two requests, namely the `"p"` and `"t"`, are both cache hits. The `"b"` is our 8th cache miss. At this point, we can evict an arbitrary choice of the `"t"`, `"a"`, or `"p"` since we won't be requesting those values again.

   The final two requests of `"z"` and `"h"` are both cache hits.

   So when all is said and done, we have a total of 8 cache misses.

2. We initially fill our cold cache with the first five requests. Because we're now dealing with an LRU cache, we fill the cache in reverse order, with the first item representing the most recently used item:

    ```
    [p, o, h, t, c]
    ```

    The next two requests, that is, the `"t"` and `"h"`, are both cache hits. We move these items to the front of our cache, which means that our cache is now:

    ```
    [h, t, p, o, c]
    ```

    The request after that, though, is our 6th cache miss since the `"z"` is not currently in the cache. We evict the `"c"` since that's the LRU item, leaving our cache as:

    ```
    [z, h, t, p, o]
    ```

    Our next request is `"o"`, which is a cache hit. We move the `"o"` to the front of the cache:

    ```
    [o, z, h, t, p]
    ```

    The next request is `"a"`, which is our 7th cache miss. We evict the `"p"`:

    ```
    [a, o, z, h, t]
    ```

    Unfortunately, we next request a `"p"`, which we literally just evicted. This is our 8th cache miss. We evict the `"t"`:

    ```
    [p, a, o, z, h]
    ```

    We next request a `"t"`, which we also evicted in the previous step. This is our 9th cache miss. After evicting the `"h"`, our cache is now:

    ```
    [t, p, a, o, z]
    ```

Our next request of `"b"` is our 10th cache miss. The cache now appears like this:

```
[b, t, p, a, o]
```

Our last two requests are `"z"` and `"h"`, which are also both cache misses. Our final cache miss tally, then, is 12.

3. The `clear_bits_2` method might raise your coworkers' eyebrows, but it does have better spatial locality. The problem with the other method, `clear_bits_1`, is that when it loads `bit_box.red_bits[i]`, the computer caches the entire array of red bits.

   However, the next step of code doesn't read from red bits; it reads from the array of *blue* bits instead! If your cache was just large enough to hold the red bits, you'll have to evict the red bits and then cache the blue bits. Once again, though, the next step of code jumps to the *green* bits, for which your cache doesn't help in any which way.

   This problem repeats itself when we start the loop again, for then the code reads from the *red* bits again, even though we only have the *green* bits in the cache.

   With `clear_bits_2`, though, once we perform our first read from the array of red bits and cache this array, our code continues to read *all* the red bits. The fact that we already have the red bits in the cache will give our code a performance boost.

   However, as we've learned, one shouldn't simply write a method like `clear_bits_2` without benchmarking it. Especially given that `clear_bits_2` is a wonky way of writing code in that it's more verbose, we'd better know for sure that it's faster. In fact, when I've benchmarked this code, I've found that `clear_bits_2` is slightly *slower* than `clear_bits_1`. This may be because, firstly, the spatial locality may not matter here since my

cache may be large enough to fit *all* the data (from red, blue, *and* green bits together). Therefore, `clear_bits_1` benefits from the cache just as `clear_bits_2` does. On top of that, `clear_bits_2` has to initiate 3 separate loops, which may slow things down slightly.

It turns out that not every algorithm that enjoys better spatial locality is faster than competing algorithms with worse spatial locality.

4. A hash table provides for O(1) searches and O(1) insertions, but doesn't easily allow us to choose a random element.

   An array, on the other hand, allows us to sample a random element in O(1) time since we can choose a random index in one step. An array can even allow us to insert new elements in O(1) time—if we append new data at the end of the array. However, an array does *not* allow for O(1) searches. Linear search of an array takes O(N) time, and even binary search (if the array is sorted) takes O(log N) time.

   So, how can we get the best of both worlds?

   Well, one approach is to *combine* a hash table *together with* an array. Here's what I mean:

   Each time we insert a new element, we insert that element into both the hash table and the array. We can insert the element as a key in the hash table, and simply make the value `True` or the like. At the same time, we also append the element to the end of the array. Yes, there's duplicate data, but the exercise didn't restrict that. In any case, we've achieved O(1) insertion.

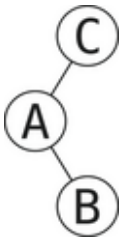
   Since the data is present in the hash table, we can also pull off O(1) searches. That is, each time we conduct a search, we always do so from the hash table, which allows us to search in O(1) time.

However, when we perform a random sample, we do so from the *array*. Again, a typical hash table doesn't allow you to pick a key at random. But since we have all the data in the array as well, we can choose a random element from the array in O(1) time.

Chapter 5

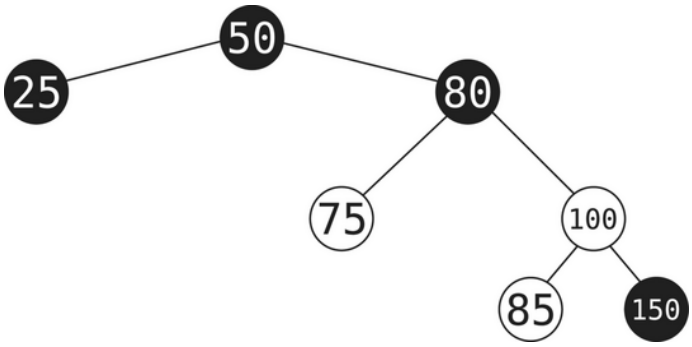
These are the solutions to the *Exercises*.

1. The tree will look like this:



   This is because we always switch the orientation of the parent-child relationship. Since previously, the A was the B's *left* child, the B will now become the A's *right* child.

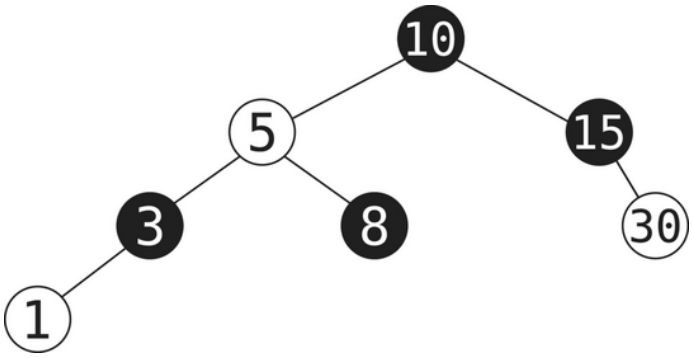
2. The red-black tree will look like this after the rotation:



   When we rotate the 80 and 100, the orientation flips, so the 100 becomes the right child of the 80. However, this creates a problem of where to place the 85. Given that the 85 was the 80's right child, where does the 85 go now that the *100* became the 80's right child?

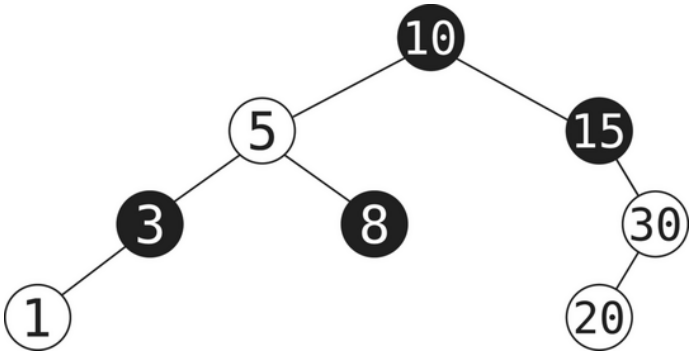
   The solution is to make the 85 a crossover node, which causes the 85 to become the 100's left child.

3. The red-black tree will look like this after the insertion:



That is, we insert a red 30 as the 15's right child. The 30 is red since all new nodes start out red. Now, given that this doesn't violate the Red Enemies Rule, there's no fixing up to do, and we can leave the tree as is.

4. Initially, we insert a 20 as the 30's left child, and color the 20 red as we do with all new nodes:



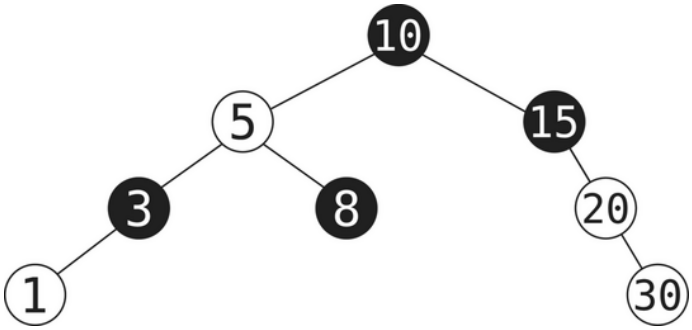
However, this violates the Red Enemies Rule because the 20 and 30, which are parent and child, are both red. This means we have some fixing up to do!

The first thing we do as part of the fixing phase is to determine whether this is a Red-Uncle Case or a Missing-Or-Black-Uncle Case. This case happens to be a Missing-Uncle Case since the 20 doesn't have an uncle. (That is, the 20's grandparent, 15, has no children other than the 20's parent, 30.)

Once we've determined that we're dealing with a Missing-Uncle Case, we next need to check whether the 20 and 30 have the same orientation

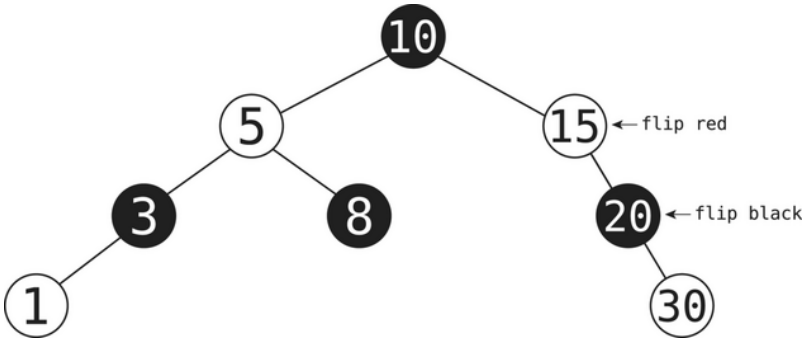
or not. Since the 20 is the 30's *left* child, and the 30 is *its* parent's *right* child, this means that they have different orientations. And *this* means that we need to execute the following steps:

First, we rotate the current node (20) and its parent (30) as shown in the tree.

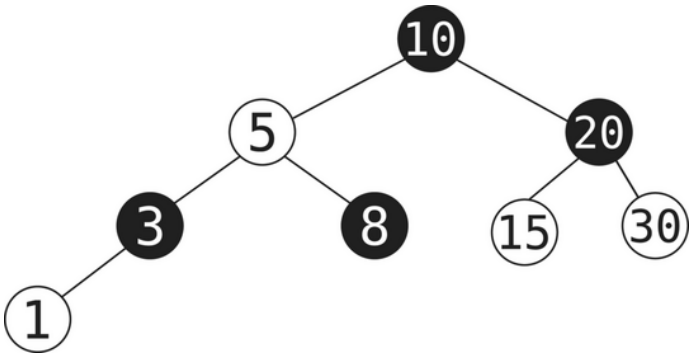


The 30 now becomes designated as the current node.

Second, we flip the current node's parent (20) black and the current node's grandparent (15) red:



The third and final step is to rotate the current node's parent (20) and grandparent (15):

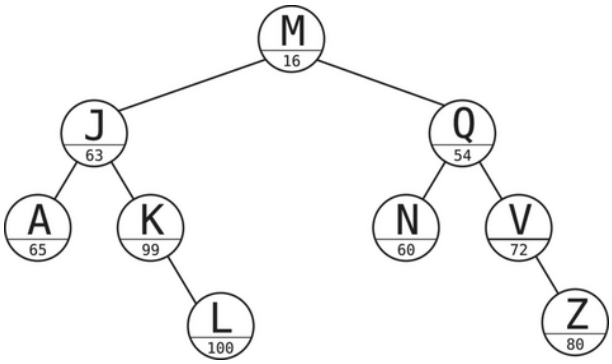


And we’re done!

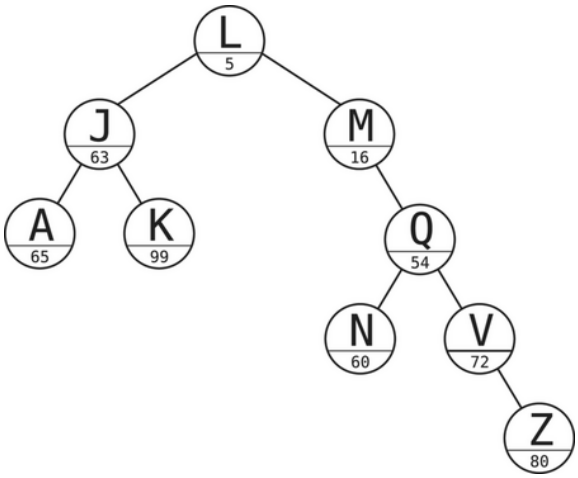
Chapter 6

These are the solutions to the *Exercises*.

1. As with a BST, we'd make the L a right child of the K. Since the L has a greater priority than its parent, the K, we don't need to enter a fixing phase. Therefore, the treap will look like the tree shown after inserting the L.

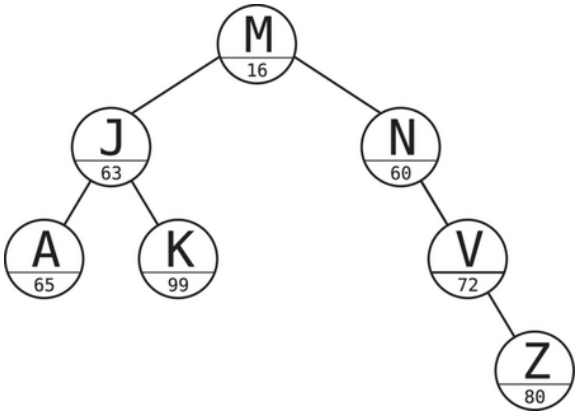


2. As with the previous exercise, we start by inserting the L as the K's right child. However, because the L has a priority that is less than any other node in the treap, we must rotate the L up the treap until it becomes the root. After all the rotations are made, the treap will look like this:



3. We delete the Q node by rotating it downward through the treap. Which child we rotate the Q with depends on which child has a lower priority than the other. In this case, the N's priority of 60 is less than the V's priority of 72, so we rotate the Q with the N.

   After this happens, the Q becomes a leaf node, and we then simply pluck it off the treap. At the end of the day, the treap will look like this after the deletion:



Chapter 7

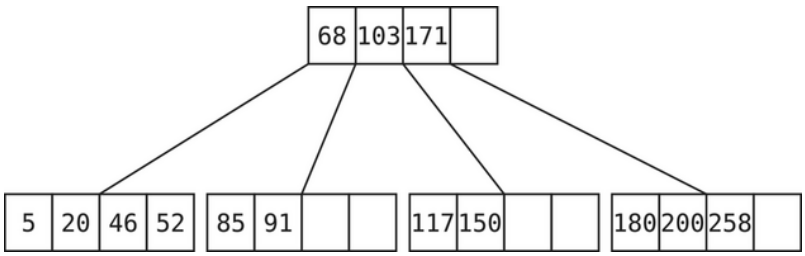
These are the solutions to the *Exercises*.

1. It takes one I/O to load each block. There are N/B blocks, which in this case is:

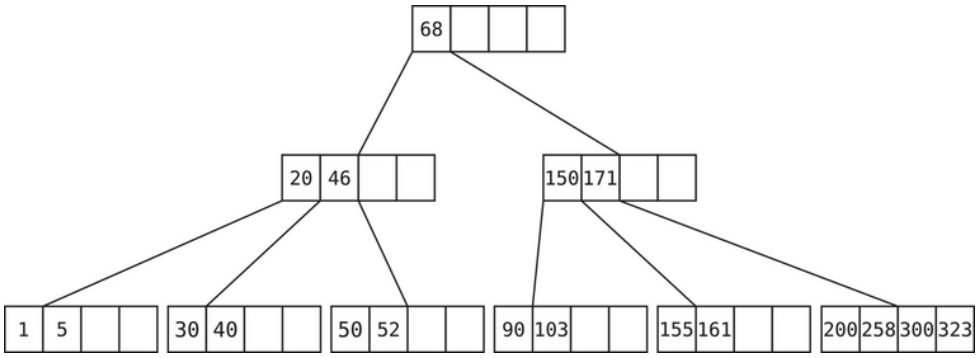
   ```
   10000 data elements / 500 block size = 2000 blocks
   ```

   And so, we have to perform 2,000 I/Os.

2. After inserting the 180, 85, 91, and 117, the B-tree will appear like this:



3. Inserting a 30, 40, and 50 will cause the tree to grow another level and end up like this:



4. Searching a B-tree takes $O(\log_{B+1} N)$ I/Os in a worst-case scenario. When a B-tree node has nodes that hold 20 values, this means that the variable B is 20 since a B-tree's node size matches the computer's block size.

In our case, we have $\log_{21}$ 100,000 I/Os, which computes to about 3.781520977582. Therefore, we can find anything in this B-tree with a maximum of 4 I/Os.

Chapter 8

These are the solutions to the *Exercises*.

1. There are a total of 128 I/Os. We can arrive at this either through analyzing the diagram or by plugging the numbers into our Big O expression. Let's do both!

   At the top level of the diagram, we can see that we take 256 items and load them into main memory. Since each block holds 4 items, this means we'll have to load a total of 64 blocks since 256/4=64. Since it takes one I/O to load one block, this means we perform 64 I/Os.

   At the next level, we take the same 256 items *again* (this time, in the form of 4 sublists that each contain 64 items) and load them into memory. As with the top level, this, too, will take 64 I/Os.

   At this point, the algorithm is complete. It comes out that we performed 64+64=128 I/Os.

   We can also plug the numbers into the Big O formula, which is O(N/B $\log_{M/B}$ N/M). In our scenario, this computes to:

   ```
   N/B             *  logM/B  N/M    =
   (256 / 4)       *  log16/4 256/16 =
   64              *     log4 16     =
   64              *  2              =
   128 I/Os
   ```

2. In this case, we can sort all 256 items in 64 I/Os since we only need to engage in one round of Mergesort. To use the Big O formula:

   ```
   N/B             * logM/B N/M =
   (256 / 4)       * (log4 256 / 64) =
   64              *  1 =
   64 I/Os
   ```

3. As we've seen, Top-Grade Merge takes O(N $\log_2$ K) time. So let's plug in our numbers:

```
N         * log2 K =
1,000,000 * log2 512 =
9,000,000
```

   Therefore, there would be a total number of 9,000,000 steps.

Chapter 9

These are the solutions to the *Exercises*.

1. Here's one way to use random sampling to determine the ratio of evens to odds:

```
import random

array = []
number_of_integers = 1000001


for i in range(number_of_integers):
    array.append(i)

random.shuffle(array)

number_of_evens = 0
number_of_odds = 0

random_sample_size = 500

for _ in range(random_sample_size):
    random_index = random.randint(0, number_of_integers - 1)
    if array[random_index] % 2 == 0:
        number_of_evens += 1
    else:
        number_of_odds += 1

percentage_of_evens = (number_of_evens / random_sample_size) * 100
percentage_of_odds = (number_of_odds / random_sample_size) * 100

print([percentage_of_evens, percentage_of_odds])
```

2. If The Formula—even once—produces a result that isn't `1`, then we know for certain that N is composite.

3. Each time I run The Formula on a prime number, there's no more than a 50 percent chance that I'll get a result of `1`. Therefore, each time I run the formula it's akin to flipping a coin. Getting a `1` four times is like a coin landing on heads four times. The odds of this are:

   ```
   1/2 * 1/2 * 1/2 * 1/2 = 1/16
   ```

   That is, there's a 1/16 chance that N in this case is prime.

Chapter 10

These are the solutions to the *Exercises*.

1. We can feed the app integers that, when divided by `997`, each produce the same remainder. For example, we can feed the app numbers that produce a remainder of `0`, such as `1994`, `2991`, `3972`, and so on. This will cause all these integers to end up in the same hash table slot, namely, index `0`. If all the integers we fed the app would have the remainder of, say, `88`, then all the integers would wind up at index `88` of the hash table.

2. Here's one way to hash strings while also incorporating randomization:

```
def hash(self, key):
    # If key is a string:
    if isinstance(key, str):
        numeric = 1


        for char in key:
            numeric *= ord(char)

        return numeric % self.prime % self.array_length

    return key % self.prime % self.array_length
```

3. So … I'm going to cover this in the next chapter. But it's good to get you thinking about this issue now, so you will better appreciate our discussion then!

Chapter 11

These are the solutions to the *Exercises*.

1. Here, I've updated the `DivisionHasher` class to incorporate the Rabin-Karp *initial* hash function as a basic way to hash any string:

```
import random


class DivisionHasher:

    def __init__(self, array_length):
        self.array_length = array_length
        self.base = 26

        # Choose a random prime number:
        p = random.randint(1000, 10000)
        while not self.is_prime(p):
            p = random.randint(1000, 10000)

        self.prime = p

    def hash(self, key):
        key = str(key)

        result = self.character_hash_code(key[0]) % self.prime

        for i in range(1, len(key)):
            result = \
              (result * self.base + self.character_hash_code(key[i]))
\
              % self.prime

        return result

    def character_hash_code(self, char):
        return ord(char) - 97

    # Fermat's Primality Test
```

```
    def is_prime(self, number):
        for _ in range(100):
            a = random.randint(1, number - 1)
            if pow(a, number - 1, number) != 1:
                return False

        return True
```

Note that in the `hash` function, I first convert the key to a string just in case it comes in as something else, such as an integer.

2. In the following code, I use the sliding window technique by maintaining a window of three characters at all times:

```
def max_vowels(string):
    num_of_window_vowels = 0

    for i in range(3):
        if is_vowel(string[i]):
            num_of_window_vowels += 1

    max_num_of_vowels_so_far = 0

    for i in range(3, len(string)):
        if is_vowel(string[i - 3]):
            num_of_window_vowels -= 1
        if is_vowel(string[i]):
            num_of_window_vowels += 1

        max_num_of_vowels_so_far = \
            max(max_num_of_vowels_so_far, num_of_window_vowels)

    return max_num_of_vowels_so_far


def is_vowel(char):
    return char in {'a': True, 'e': True, 'i': True, 'o': True, 'u':
True}
```

I begin by creating the initial window and counting how many vowels it contains. We store this number inside the variable

`num_of_window_vowels`. Then, we move the sliding window along using a `for` loop that runs from index `3` until the end of the `string`. Each time we shift the window forward, we check to see if we dropped a vowel, in which case we decrement the `num_of_window_vowels` by 1. We also check to see if the window *gained* a vowel on its right end, in which case we increment `num_of_window_vowels` by 1.

We track the greatest number of vowels in any window within the `max_num_of_vowels_so_far` variable, which is what we return at the end of the function.

3. First, I'll show you the code, and then I'll explain it:

```
def length_of_longest_substring(string):
    if not string:
        return 0

    left = 0
    right = 0
    max_distance_so_far = 0
    current_window_chars = {}

    while right < len(string):
        if string[right] not in current_window_chars:
            current_window_chars[string[right]] = True
            max_distance_so_far = max(max_distance_so_far, right -
left)
            right += 1
        else:
            del current_window_chars[string[left]]
            left += 1

    return max_distance_so_far + 1
```

This code features a sliding window, but the size of the window expands and contracts as needed. To allow our window to change size, we establish *two* pointers, with `left` pointing to the left-most index of the window, and `right` pointing to the right-most index of the window.

Let me walk through what the code does using an example.

Say that our input string is `"abac"`. The `left` and `right` pointers start out both pointing to the first character, `"a"`. Throughout our algorithm, we keep track of the `max_distance_so_far`, which is the greatest distance between the two pointers. At the beginning, since both pointers are at the same spot, their distance is `0`.

We also establish a hash table called `current_window_chars`, which keeps track of which characters are contained within the current window. If a character is inside the window, the hash table will contain that character as a key. (I made `True` the arbitrary value associated with each key.)

The guts of the algorithm is a `while` loop that lasts until the `right` pointer reaches the end of the string. In this loop, we move the right pointer to the right, one character at a time. As we do, we *expand* the window, since the `left` pointer is standing still. With each new character that enters our window, we insert that character into the `current_window_chars` hash table.

After one round of the loop with our example string `"abac"`, `left` points to the first `"a"` and `right` points to the `"b"`. The distance between the two pointers is `1`, and given that this is the maximum distance we've encountered so far, this becomes the new `max_distance_so_far`.

In each round of the loop, we check the hash table to ensure that the new character we include in the window isn't already inside the hash table.

In the next round of the loop, the `right` pointer encounters the *second* `"a"`. This is an "invalid" window since it contains two instances of the

character `"a"`. That is, `right` points to the second `"a"` while `left` still points to the first `"a"`.

Now, we might be tempted to have the `left` pointer skip to where the `right` pointer currently is and continue with the loop. However, this would be a mistake since in truth, the `"b"` is part of the longest valid window, `"bac"`. So instead, we move the `left` pointer along, which thereby *shrinks* the window since now `left` has moved closer to `right`. Because the window shrinks, we also delete the old character `left` was pointing to from our hash table. From this point, we continue once again to move the `right` pointer along and expand the window once again.

In our example, the window will eventually encompass the characters `"bac"`.

The loop finally terminates once `right` hits the end of the input string. At this point, `max_distance_so_far` will be the greatest distance between the two pointers we've ever encountered. In truth, though, the *length* of the largest window is the greatest *distance* plus 1. That is, in `"bac"`, with `left` pointing to `"b"` and `right` pointing to `"c"`, the distance between the two pointers is 2. However, the actual *length* of the substring is *3*, so that's the number our function finally returns.

Chapter 12

These are the solutions to the *Exercises*.

1. Here is one way we can convert a binary string into a decimal number:

```
def decimal(string):
    integer = 0
    power = 0
    index = len(string) - 1

    while index >= 0:
        if string[index] == "1":
            integer += 2**power

        index -= 1
        power += 1

    return integer
```

   This function accepts a `string` parameter. We expect this to be a binary string such as `"000101001010"`.

   We start by initializing an `integer` at `0`. Eventually, this will be the decimal number we return at the end of our function.

   We also initialize a `power` variable, which we'll use to help us compute what number each digit place of our binary string represents. The right-most digit place will be `2` to the `power` of `0` (that is, the ones place). The next digit place to the right will be `2` to the `power` of `1`, which is the twos place. The digit place immediately to the right of that is `2` to the `power` of `2`, which is the fours place, and so on.

   We then begin a loop which scans our binary `string` from right to left by tracking an `index`. We use the current `power` to compute the number that is represented by the digit place of the current `index`. If there is a `1` at

the current digit place, we take the number represented by the current digit place and add it to `integer`. For example, if we're looking at the eights place and there's a `1` bit there, we add `8` to `integer`. If there's a `0` bit there, we simply proceed to the next round of the loop without modifying `integer` in the current round.

Alternatively, there's another way we could have computed the value of each digit place without tracking a `power` variable. Instead, we could compute each digit place by doubling whatever the previous digit place to the right represented:

```
def decimal(string):
    integer = 0
    place = 1
    index = len(string) - 1

    while index >= 0:
        if string[index] == "1":
            integer += place

        index -= 1
        place *= 2

    return integer
```

When I benchmark both approaches, this second version turns out to be faster. This is because computing powers is slower than performing simple multiplication.

2. Here is a Python-based approach for converting a decimal number into a binary string:

```
def binary(number):
    place = 2147483648
    binary_string = ""

    while place >= 1:
        if number >= place:
```

```
            binary_string += "1"
            number -= place

        else:
            binary_string += "0"

        place //= 2

    return binary_string
```

This function accepts a `number` parameter; this is the decimal number we will convert into a binary string.

The first thing that will undoubtedly jump out at you is the seemingly random integer of `2147483648` that we initialize our `place` variable with. However, I've chosen this number with careful precision. Because the exercise asks us to return a string containing exactly 32 bits, this means that the right-most bit represents the `2147483648`s place. That is, `2` to the power of 31 is `2147483648`. (Remember, it's the *second-to-right-most* bit that represents `2` to the first power. Accordingly, 30 bits to the right of that will represent 2 to the 31st power.)

We initialize a `binary_string` as an empty string. By the time our function is complete, this will contain 32 characters that are either `"0"` or `"1"`, such as `"00000000000000000110100110101011"`. So, this is what we'll return at the end of our function.

In the meantime, though, we begin a `while` loop. In the loop's first round, we check to see if there should be a `1` in the `2147483648` place. This would be the case if our input `number` is greater than or equal to `2147483648`. If we find that `number` is indeed greater than or equal to `2147483648`, we place a `"1"` in that spot of the `binary_string`.

As an example, let's pretend our input `number` is a smaller number, such as `9`. To determine if we should place a `1` bit in the eights place, it

all depends on whether `9` is greater than or equal to `8`. Since `9` is greater than `8`, we'll put a `1` bit in the eights place. If we put a `0` bit in the eights place, it's impossible to produce a binary string that can equal `9` with the remaining digit places to the right. That is, the greatest number we can represent with a `0` bit in the eights place is `0111`, which is only `7`.

If our `number` is *smaller* than `8`, though, we certainly can't put a `1` bit in the eights place, since our binary string would then represent a number of `8` or greater.

In any case, if within a particular loop round we do place a `"1"` into our `binary_string`, we then reduce `number` by the `place` we're in to determine what the rest of the binary string should look like. Again, dealing with the example `number` of `9`, once we place a `1` bit in the eights place, this means that we need to figure out how to represent what *remains* of `number`, specifically `1`, as `9 - 8 = 1`.

Whether we place a `1` bit or `0` bit in the current place, we then compute the next place to the right by halving `place`. By the time our loop is complete, `binary_string` will contain some combination of 32 `0` and `1` bits that properly represent our input `number`.

3. Here, I've added three methods to our `BitVector` class:

```
def union(self, other_bit_vector):
    bv = BitVector(self.range_of_bits)

    for i in range(len(self.integers)):
        bv.integers[i] = self.integers[i] |
other_bit_vector.integers[i]

    return bv

def intersection(self, other_bit_vector):
```

```
        bv = BitVector(self.range_of_bits)

        for i in range(len(self.integers)):
            bv.integers[i] = self.integers[i] &
    other_bit_vector.integers[i]

        return bv

    def difference(self, other_bit_vector):
        bv = BitVector(self.range_of_bits)

        for i in range(len(self.integers)):
            bv.integers[i] = self.integers[i] &
    ~other_bit_vector.integers[i]

        return bv
```

The `union` method uses OR as I described in the chapter. However, instead of ORing two integers, we OR two *arrays* of integers. To do this, we run a loop in which we iterate over each index (`i`) of both arrays. When `i` is `0`, for example, this means we OR the first integer of the first bit vector with the first integer of the second bit vector. We place the result in a brand-new third bit vector's underlying array. Finally, we return the new bit vector.

The `intersection` and `difference` methods work similarly, except that they use their appropriate bitwise operators.

4. We *could* compute the hamming distance by scanning the two integers and comparing the integers' bits at each digit place. But why do all that work when our good pal XOR can do the heavy lifting for us?

   As I mentioned in the chapter, XOR is essentially a litmus test that reveals precisely where two integers have differing bits. Specifically, it does this by placing a `1` bit in each digit place where the two integers have opposite bits.

So, our approach is to XOR the two integers (`x` and `y` in my code that follows), producing an integer called `difference`, and then we count how many `1` bits are in `difference`:

```
def hamming_distance(x, y):
    difference = x ^ y

    bit_count = 0

    for n in range(0, 32):
        mask = 1 << n
        if mask & difference != 0:
            bit_count += 1

    return bit_count
```

The `for` loop here counts the `1` bits by using the same `mask` technique used in the `read_bit` method of our `BitVector` class. And so, we read each of the 32 bits of `difference` and count up the `1` bits.

5. Did I mention that this is one of my favorite puzzles? The solution isn't obvious, but it's super fun. Let me break it down piece by piece.

   The hero of this solution is our good pal XOR. Let's talk about XOR a bit more.

   One thing to highlight about XOR is that when we XOR two identical integers, the result will be `0`. Think again about XOR being that litmus test for revealing `1` bits wherever two integers have differing bits. It emerges that if we XOR two identical integers (such as `5` and `5`), XOR will produce only `0` bits. And, as you know, an integer with only `0` bits is, well, `0`.

   So, let's pretend for a moment that our input array is `[3, 3, 7, 7, 4, 4, 1]`. As you can see, this array has two instances of `3`, `7`, and `4`; however, there's only one instance of `1`. Let's XOR all these numbers up.

When we XOR the two `3`s, we get `0` since the `3`s are identical. When we take this resulting `0` and XOR it by the `7`, we'll get some nonzero result. But it's not going to matter because when we XOR that result by the second `7`, the result will be `0` again. The same happens for the two `4`s.

By the time we get up to the `1`, our result will be `0`. When we XOR the `0` with `1`, we'll get `1` since one of the rules of XOR is that when we XOR `0` with some other number, the result will be that other number.

It turns out that the final result of XORing all the numbers will be *the very number we're seeking*—that is, the number that only appears once in the array.

Here's the surprisingly concise solution code:

```
def single_number(array):
    running_total = 0

    for num in array:
        running_total ^= num

    return running_total
```

Now, you're probably wondering, "Well, that works if the numbers are ordered that way, where each pair of integers appears together, and the single number is located at the end. But what if the integers aren't sorted in any particular order?"

That's a great question; I had it too. The answer, though, is that when you XOR a bunch of numbers together, you'll always get the same result *no matter the order of the numbers*. In fancy terms, this is the *commutative property*. Just as the order of numbers doesn't matter when you add them or multiply them together, the same applies to XORing.

This isn’t necessarily intuitive, but try it out for yourself and you’ll see it’s true!

So, at the end of the day, the result of XORing all the array’s numbers will be the number we’re looking for—namely, the integer that appears only once in the array. This makes for a great party trick at a nerdy event (but not *too* nerdy, since you don’t want everyone there to already know the solution before you show them).

Chapter 13

These are the solutions to the *Exercises*.

1. You can store all usernames inside an in-memory Bloom filter. Then, each time a user asks if a particular username is available, your app would check for its existence inside the Bloom filter. Bloom filter lookups have the potential to be a lot faster than database lookups.

   Because a Bloom filter never produces a false negative, it can be trusted to say if a given username is available. (That is, it doesn't exist within the Bloom filter.) Accordingly, we never have to worry that we'll inadvertently allow two users to end up with the same username.

   At the same time, a Bloom filter can produce false positives, which means that it's possible that the Bloom filter may tell us that a username is unavailable even though it's not taken. However, this may not be a big deal, since the user can find some other username (that's equally unclever) to use.

2. Luckily for HumbleBrag, you put the kibosh on the engineering team's plan. Since a Bloom filter can produce a false positive, it's possible that the Bloom filter will validate the entered password even if it's not correct! If ten strings hash into the same set of bits in the Bloom filter, any one of those strings would be accepted as the correct password, even though only one of those passwords is the right one.

3. With this code, we can get the info we need:

```
import sys
import bloom_filter

bf = bloom_filter.BloomFilter(1000, 0.03)
print(bf.m)  # number of bits
print(bf.k)  # number of hash functions
print(sys.getsizeof(bf.bv.integers))  # bit vector's number of bytes
```

When I run this, I find that M (the number of bits) is 7,298, and that K (the number of hash functions) is 5.

Now, I'd expect that if the Bloom filter takes up 7,298 bits, then it takes up around 912 bytes since `7298 // 8 = 912`. However, when I run `sys.getsizeof()` on the actual underlying array of the bit vector (which, in turn, underlies the Bloom filter), I find that it takes up 1,888 bytes. This, again, is because Python crams some extra info into its arrays and integers. So, in reality, the size of the Bloom filter is somewhat larger than the strict computation of M.

Thank you!
We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head on over to https://pragprog.com right now, and use the coupon code BUYANOTHER2025 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With up to a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-author/ today to learn more and to get started.

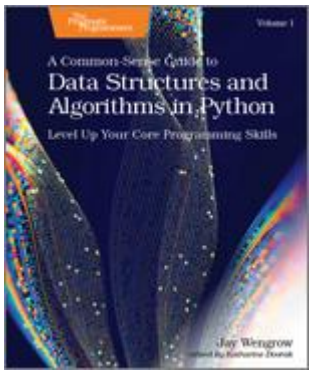
Thank you for your continued support. We hope to hear from you again soon!

The Pragmatic Bookshelf

You May Be Interested In…

Select a cover for more information

A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1



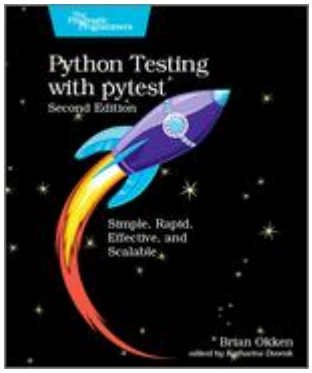
If you thought data structures and algorithms were all just theory, you're missing out on what they can do for your Python code. Learn to use Big O notation to make your code run faster by orders of magnitude. Choose from data structures such as hash tables, trees, and graphs to increase your code's efficiency exponentially. With simple language and clear diagrams, this book makes this complex topic accessible, no matter your background. Every chapter features practice exercises to give you the hands-on information you need to master data structures and algorithms for your day-to-day work.

Jay Wengrow

(502 pages) ISBN: 9798888650356 $57.95

Python Testing with pytest, Second Edition

Test applications, packages, and libraries large and small with pytest, Python's most powerful testing framework. pytest helps you write tests quickly and keep them readable and maintainable. In this fully revised edition, explore pytest's superpowers—simple asserts, fixtures, parametrization, markers, and plugins—while creating simple tests and test suites against a small database application. Using a robust yet simple

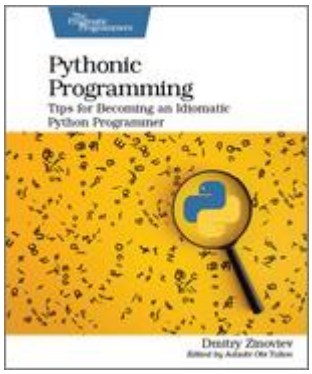


fixture model, it's just as easy to write small tests with pytest as it is to scale up to complex functional testing. This book shows you how.

Brian Okken

(272 pages) ISBN: 9781680508604 $45.95

Pythonic Programming



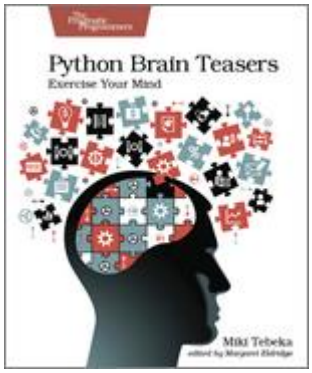
Make your good Python code even better by following proven and effective pythonic programming tips. Avoid logical errors that usually go undetected by Python linters and code formatters, such as frequent data look-ups in long lists, improper use of local and global variables, and mishandled user input. Discover rare language features, like rational numbers, set comprehensions, counters, and pickling, that may boost your productivity. Discover how to apply general programming patterns, including caching, in your Python code. Become a better-than-average Python programmer, and develop self-documented, maintainable, easy-to-understand programs that are fast to run and hard to break.

Dmitry Zinoviev

(150 pages) ISBN: 9781680508611 $26.95

Python Brain Teasers

We geeks love puzzles and solving them. The Python programming language is a simple one, but like all other languages it has quirks. This

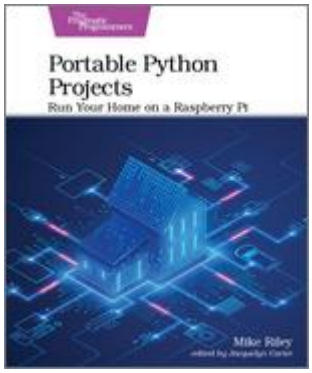


book uses those quirks as teaching opportunities via 30 simple Python programs that challenge your understanding of Python. The teasers will help you avoid mistakes, see gaps in your knowledge, and become better at what you do. Use these teasers to impress your co-workers or just to pass the time in those boring meetings. Teasers are fun!

Miki Tebeka

(116 pages) ISBN: 9781680509007 $18.95

Portable Python Projects

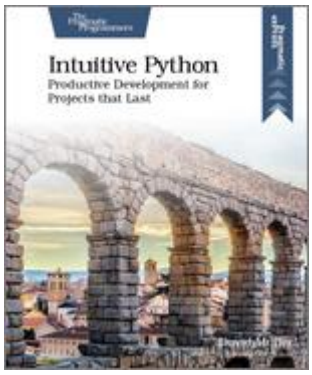


Discover easy ways to control your home with the powerful new Raspberry Pi hardware. Program short Python scripts that will detect changes in your home and react with the instructions you code. Use new add-on accessories to monitor a variety of measurements, from light intensity and temperature to motion detection and water leakage. Expand the base projects with your own custom additions to perfectly match your own home setup. Most projects in the book can be completed in under an hour, giving you more time to enjoy and tweak your autonomous creations. No breadboard or electronics knowledge required!

Mike Riley

(180 pages) ISBN: 9781680508598 $45.95

Intuitive Python

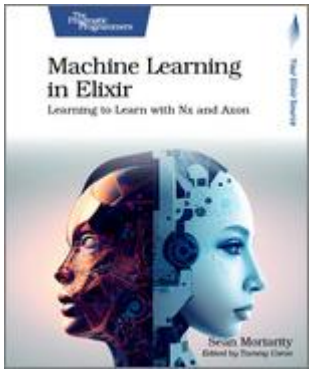


Developers power their projects with Python because it emphasizes readability, ease of use, and access to a meticulously maintained set of packages and tools. The language itself continues to improve with every release: writing in Python is full of possibility. But to maintain a successful Python project, you need to know more than just the language. You need tooling and instincts to help you make the most out of what's available to you. Use this book as your guide to help you hone your skills and sculpt a Python project that can stand the test of time.

David Muller

(140 pages) ISBN: 9781680508239 $26.95

Machine Learning in Elixir

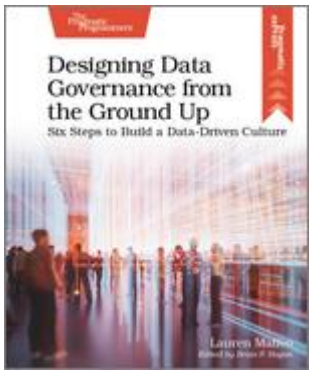


Stable Diffusion, ChatGPT, Whisper—these are just a few examples of incredible applications powered by developments in machine learning. Despite the ubiquity of machine learning applications running in production, there are only a few viable language choices for data science and machine learning tasks. Elixir's Nx project seeks to change that. With Nx, you can leverage the power of machine learning in your applications, using the battle-tested Erlang VM in a pragmatic language like Elixir. In this book, you'll learn how to leverage Elixir and the Nx ecosystem to solve real-world problems in computer vision, natural language processing, and more.

Sean Moriarity

(372 pages) ISBN: 9798888650349 $61.95

Designing Data Governance from the Ground Up



Businesses own more data than ever before, but it's of no value if you don't know how to use it. Data governance manages the people, processes, and strategy needed for deploying data projects to production. But doing it well is far from easy: Less than one fourth of business leaders say their organizations are data driven. In Designing Data Governance from the Ground Up, you'll build a cross-functional strategy to create roadmaps and stewardship for data-focused projects, embed data governance into your engineering practice, and put processes in place to monitor data after deployment.

Lauren Maffeo

(100 pages) ISBN: 9781680509809 $29.95
