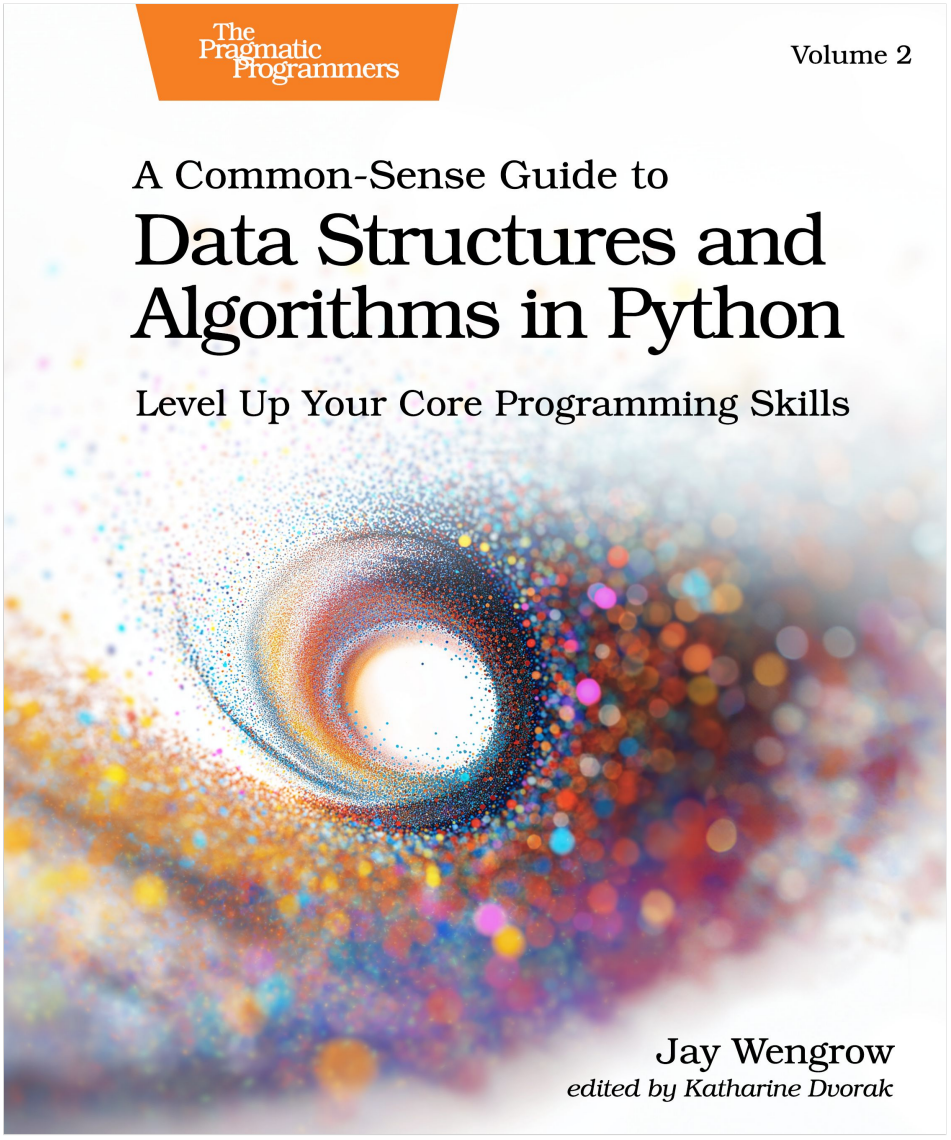


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

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, a tree specialized for storing *lots* of information.

In this as well as the next chapter, we 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 just 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 chose, you could skip ahead at this point to Chapter 9 and return to Chapters 7 and 8 at some later time.

External Memory

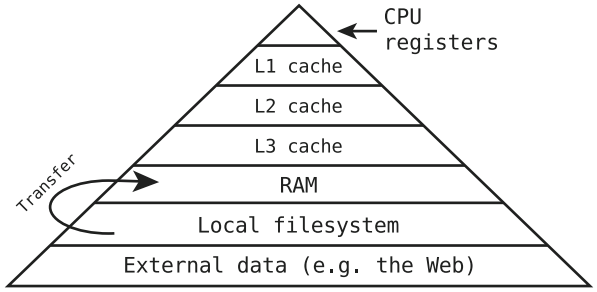
Let's get back to our 50GB list of integers. To store such a list, we need to store the integers somewhere *outside* main memory, since the list certainly won't fit *inside* main memory. Typically, we'd do this in the filesystem, assuming that our filesystem *is* larger than 50GB.

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 not does not deal directly with data from the filesystem. We touched on this earlier in The Memory Hierarchy, on page ?, but let me review these details and elaborate on them a bit further, as these concepts are the foundation of the 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 actually do something with that data.

Using the memory hierarchy diagram you first saw on page ?, we're essentially referring to the following transfer of data:



In truth, some of the data may also be copied into the various cache levels. And so when I refer to either "main memory," "RAM," or even simply "memory" in this chapter, I mean to include the cache levels as well.

And 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 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:

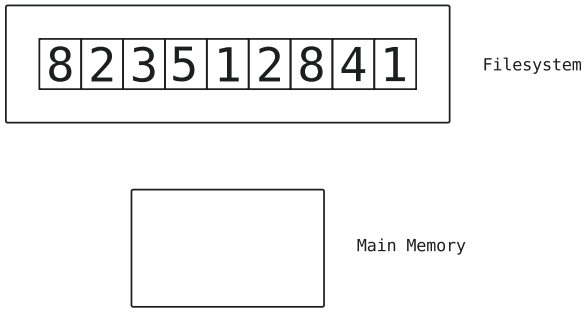
Because our computer has 8GB RAM, the computer will start out by loading the first 8 blocks of data from our 50GB file, with each block containing 1GB of data. This will fill the capacity of our RAM, since our RAM is 8GB, and the 8 blocks contain a combined total of 8GB of data. Our algorithm will then 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 actu-

ally 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, now, 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.

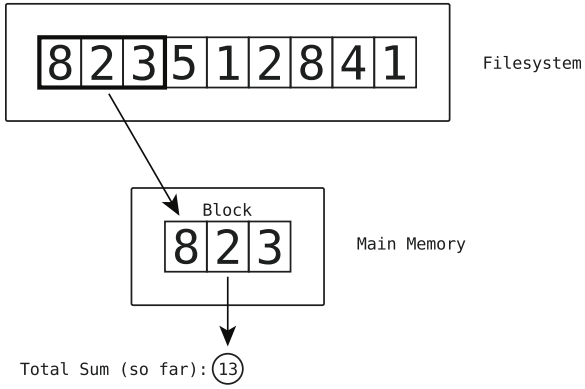
Here's a depiction of our computer's main memory, and a file in the filesystem containing integers that we want to sum up:



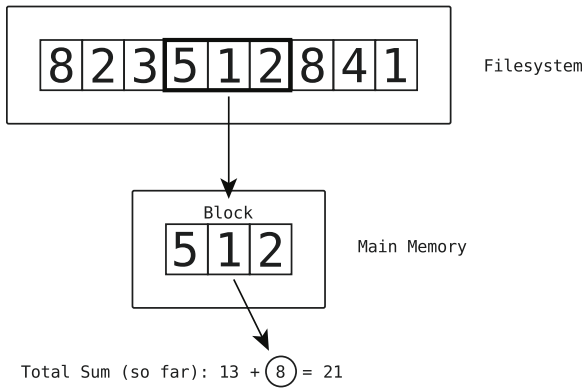
Currently, the main memory is empty.

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.

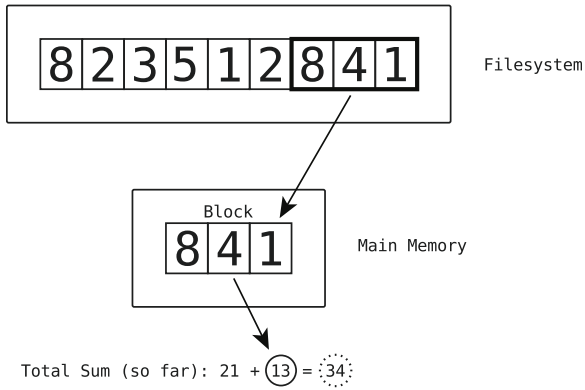
So we first have to load a block of 3 integers from the file, and compute the sum of those 3 integers:



We then load the second block from the file, sum *those* integers, and add that to our total sum so far:



Finally, we do the same with the last block:



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 really 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.

So 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 see soon 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 really much slower than just 50 steps. It's more like 1,049 steps! There's 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 just need to modify the way we *use* Big O notation to accommodate external-memory algorithms.