INTRODUCTION

This is a book about computational thinking through the lens of *data structures*, constructs for organizing and storing data. It is more than a cookbook of handy data structures. Rather, it explores the thinking behind these structures and their fundamental impact on solving complex problems, using real-world analogies to make abstract computational concepts intuitive. The goal of this book is to provide new insights into how you can use preexisting structure within the data to your advantage or create new structures to efficiently solve problems.

Among other things, I discuss the differences between arrays and linked lists, the complexity and power of pointers, the effect of data structures on algorithmic behavior, the branching of tree-based data structures, mathematical mappings in hash tables, and the usefulness of randomization. In short, you'll learn to think about algorithms by investigating different ways to organize the data they process. You'll also apply these computational approaches to real-world problems, a surprising number of which focus on procuring a decent cup of coffee.

Understanding how data structures function is critical to using them effectively. Just as an experienced carpenter wouldn't pound screws into wood with a hammer or use sandpaper to cut a two-by-four in half, an experienced programmer needs to choose the right tools for every job. As we'll see repeatedly throughout the following chapters, every data structure comes with tradeoffs. Saws cut through wood more effectively than sandpaper but create coarse edges. There is no single data structure that is perfect for every possible use case, but this is what makes computer science and the development of algorithms so interesting. A good computer scientist must understand how different data structures behave in order to determine where they can be best used.

This book focuses on a few canonical data structures and uses them to explore fundamental themes in computational thinking. Each of these data structures is a useful exemplar of a more general class of data structures and of a conceptual approach. For example, B-trees demonstrate one approach to the problems of keeping search trees balanced and optimizing for expensive memory accesses. I discuss the tradeoffs between memory usage and accuracy with Bloom filters; the use of randomization with skip lists; and how to capture multidimensional structure with grids, quadtrees, or k-d trees. As such, this book is neither an introduction to programming, a comprehensive anthology of data structures, nor a full analysis of brewing coffee (although we will touch repeatedly on this important topic). Our goals are different—to develop mental tools that apply across a range of specific problems and programming languages.

Intended Audience

This book is for anyone who wants to learn more about the thinking behind the data structures that lie at the heart of computer science. I assume such basic familiarity with programming as can be expected after taking an introductory course, participating in a boot camp, or working through a beginners' programming book. Readers should be familiar with fundamental programming concepts such as variables, loops, and conditional statements. Some more adventurous readers might even have coded up some of the data structures or algorithms in this book already or might do so as they read through it. However, you won't need to know the specific details of particular programming languages or algorithms.

I hope this book appeals to a wide range of audiences. Practitioners who have learned basic programming from an introductory course will find an introduction to computational thinking that can provide a foundation for future investigation. Students will find a new way to understand particularly difficult or tricky topics. Mathematicians will learn new names and jargon for ideas they've used since well before computer science existed. And experienced computer scientists will find amusing new analogies to explain concepts they use every day.

Language-Agnostic

This book is designed to apply to a wide range of programming languages. While this might come as a disappointment to the more opinionated readers who want to either (a) see their favorite language featured throughout these pages or (b) argue about the author's terrible language preferences and how they must reflect suboptimal life choices (since programming languages, like sports teams, are always a topic for heated debate), the concepts presented in the book are generally applicable across a range of languages. You can implement a binary search tree in almost any language, for instance. In fact, most programming languages already include many of these basic data structures as part of their core language or a standard library.

The book uses pseudocode examples that are largely based on Python's general syntax, since Python is a widely used and easily readable programming language. I denote code blocks via indentation, use standard equality notation (`==` for equal and `!=` for not equal), use `True` and `False` to indicate Boolean values, denote comments with lines starting with the `#` symbol, and pass composite data structures as references. Arrays are zero indexed, and the value at index `i` is referenced as `arr[i]`.

However, I also deviate from the Python syntax wherever this aids readability. I specify all variables as `Type: Name` to make the types explicit, and I use the value `null` to indicate a null pointer. I often use `WHILE` loops over `FOR` loops or other compact forms to clearly show how the loop is iterating and its termination condition.

I've intentionally kept the examples in this book simple in order to focus on the computational ideas behind them. This means that individual implementations may not be fully optimized and will often be more verbose than strictly necessary. Throughout the text, I break out different conditions to illustrate the thought process behind the approach. At times the implementations vary from programming best practices in order to structure the code in a way that matches the explanation. In addition, to keep the examples simple, I often leave out the basic validity checks that are vital to include in production programs, such as checking that our array access is inbounds. Needless to say, treat these examples only as illustrations of the concepts at hand, rather than using them verbatim in your own projects. This is a good rule in general: never treat pseudocode as a full implementation. Always incorporate the relevant testing, validity checks, and other best practices when implementing algorithms yourself.

On Analogies and Brewing Coffee

This book makes extensive use of metaphor and analogy, illustrating complex technical concepts by comparison to (sometimes absurd) real-world scenarios. Similes are scattered through this book like blueberries through a muffin. Each chapter explains the intricate working of data structures and algorithms with examples ranging from organizing your kitchen to determining whether you've ever tried a specific brew of coffee, inviting you to consider how these computational concepts work in a different way from computer code.

The examples will often bend the rules of reality, be oversimplified, or border on the ridiculous. For example, we repeatedly consider the question of storing and sorting extensive coffee collections, ignoring the tragic fact that coffee does go stale. While that means this book is not a strictly realistic guide to making the ultimate cup of coffee, the absurd analogies keep things fun and should encourage you to think outside your normal approaches. Simplifying the analogies allows us to focus on just those aspects that are critical to the computational concept. For example, when discussing the use of nearest-neighbor search to find a close cup of coffee, I focus on distances (the core computational concept) and neglect such complicating factors as fences or rivers. My goal is to tailor the analogy to the core of the problem.

I use analogies to augment formal descriptions and precise code. Personally, I find it easier to break free from technicalities and minutiae when viewing a data structure's operation in an active, narrative context with people (or even overcaffeinated squirrels) interacting with physical objects, rather than sticking to the vocabulary of `FOR` loops and variable counters. Visualizing a frantic chase through a maze of alleyways provides a different perspective of a graph algorithm from the formal context of iterating over abstract nodes and edges. I encourage readers to map these analogies to their own broad range of concepts, whether part of their daily life or fancies of the absurd.

How to Use This Book

The book is structured progressively. That is, while each chapter focuses on a different computational concept—either a data structure or motivating problem—each also builds upon the previous chapters. Most of the later chapters, for example, rely on the discussion of memory-linked data structures and the use of pointers introduced in Chapter 3. We return to the basic binary search tree structure introduced in Chapter 5 again and again as we examine variations of branching data structures. Thus, I recommend that you approach the chapters in order.

As we explore different data structures and how they apply to various problems, we will see consistent themes appear, including:

- The impact of the data's structure on algorithms accessing it
- How to think about performance in the worst case
- The importance of allowing for dynamic changes in your data set and how to efficiently enable these changes
- Tradeoffs among memory, runtime, code complexity, and accuracy
- How we may need to tune data structures for the problem and what tradeoffs to consider
- How we can adapt data structures to tackle new problems

These themes provide both a framework for thinking about the data structures and a set of questions to ask when facing a new problem. A critical aspect of choosing the data structure is understanding why it performs the way it does and how it will apply to new data.

Most of all, the two questions that you should keep in mind throughout the book are "How?" and "Why?" *How* does a given data structure enable a computation? *How* do we structure the data to maximize efficiency in a given context? *Why* does a given structure enable these computations? *How* does this data structure break down in a different context? *Why* is the author using that ridiculous analogy? *Why* is the author so obsessed with coffee? Understanding the answers to these questions (other than the last one) will provide the foundation you need to effectively use already-existing data structures and develop novel techniques in the future.

1
INFORMATION IN MEMORY

Any remotely interesting computer program needs to be able to store and access data from memory. This data might be the text in a document, the information on a web page, or the details of every variety of coffee we've ever sampled stored within a database. In each case, the data is fundamental to the program performing its intended function.

These examples only represent the data that users see and think about. The program must also track numerous pieces of data behind the scenes, such as how many times we have passed through a loop, the current location of our character in a game, or the current system time. Without this data, a program can't represent changes to its internal state.

In this chapter, we examine the very basics of storing data in memory. We'll look at how the simplest data structures—plain old variables, composite data structures, and arrays—store their data. We'll also introduce the book's pseudocode conventions. For readers who have experience programming, this chapter's key concepts might already be familiar. Even so, they're a critical starting point for our journey and worth a review since they provide the foundations to build more powerful and exciting data structures.

Variables

Individual pieces of data are often stored in *variables*, which are essentially names representing the location (or *address*) of a piece of data in the computer's memory. Readers with even a passing exposure to programming will already be familiar with variables: they are a foundational concept in computer science, necessary for even the simplest programs. Variables enable programs to track information that changes throughout the course of the program. Need to count how many times you have passed through a `FOR` loop? Track the player's score in a game? Count how many spelling errors you've made while writing an introductory chapter about variables? Use a variable.

Without variables, a programmer can't track, evaluate, or update the program's internal state. When you create a variable, the system allocates and assigns it a location behind the scenes. You are then free to write data to that location using a variable name of your choice and look it up using the same name. As long as you know the variable's name, you don't need to know the memory location of the data. We can visualize the computer's memory as a long column of bins. Each variable occupies one or more contiguous bins, depending on the size of the variable, as shown in *Figure 1-1* for three variables: `Level`, `Score`, and `AveScore`. In this illustration, the average score (`AveScore`) is a floating-point number (a number with a decimal) that uses two bins of memory.

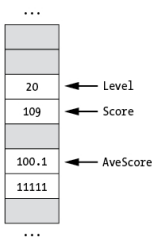


Figure 1-1: Computer memory depicted as a column of bins

In some ways, variables are like the little paper labels on file folders, like the ones in *Figure 1-2*: once we have attached the label, we don't need to remember the folders' order or exactly how we stored them. We just look up the folder by its label—but this means that it's important to use informative names. The author's own filing cabinet is crammed full of overloaded folders with names such as *Stuff*, *Misc*, *Important*, and *Other Stuff*, making it difficult to know what is stored inside. Likewise, vague variable names make it hard to guess what values they represent.

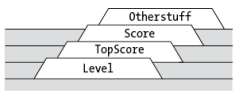


Figure 1-2: Variables, like the labels on file folders, provide a convenient way to find and access your stored values.

In many programming languages, variables have an associated type that denotes exactly what type of data they store, such as integers, "floats" for floating-point values, or Booleans for true or false values. These types tell the program how much memory the variable occupies and how to use it. A Boolean variable, for example, stores a limited range of values and often requires only a small amount of memory. A double-precision floating-point number might store a much larger and more precise number and so would use multiple bins. The syntax of defining types, and even whether types need to be explicitly defined, varies among programming languages.

Throughout this book, we will use the language-independent `<type>: <name>` pseudocode format to specify our variables in examples. For example:

```
Integer: coffee_count = 5
Float: percentage_words_spelled_correctly = 21.0
Boolean: had_enough_coffee = False
```

Sometimes a variable will be of the general type `Type` to indicate that it could take on a range of types depending on the implementation. We'll operate on the variables using syntax typical of most programming languages, including the use of `=` for assignment:

```
coffee_count = coffee_count + 1
```

For numeric types, including integers and floats, we'll use standard arithmetic operations, such as `+`, `-`, `*`, and `/`. For Boolean data types, we'll use Boolean operations, such as `AND`, `OR`, and `NOT`. The syntax you'll need to use in your programs will vary depending on your programming language (and is a common focal point in fights over the relative merits of different languages).

Composite Data Structures

Many programming languages provide the ability to create *composite data structures*, such as a struct or an object, which gather multiple individual variables into a single group. Composite data structures provide an easy way to gather related pieces of data and pass them around together. For example, we might define a `CoffeeRecord` to track some information about the kinds of coffees we have sampled:

```
CoffeeRecord {
    String: Name
    String: Brand
    Integer: Rating
    Float: Cost_Per_Pound
    Boolean: Is_Dark_Roast
    String: Other_Notes
}
```

Instead of maintaining six individual variables to track a coffee's properties, we store all of that information in a single composite data structure, `CoffeeRecord`. Of course, a true coffee connoisseur would likely track a few hundred additional properties, as well as exact information about the date, time, location, and weather conditions related to the coffee consumption. Coffee is, after all, a complex subject and deserves thorough documentation. Each additional property further underscores the importance of using a composite data structure: the alternative of passing around hundreds of related variables not only is tedious but also increases the probability that the programmer will make a mistake, such as passing variables to a function in the wrong order.

Business cards provide a real-world example of composite data structures. Each individual card is a packet of data containing multiple pieces of information such as your name, phone number, and email address. Bundling this information into a single card increases the efficiency of tracking it and passing it around. Imagine the mess and confusion of handing a colleague five different scraps of paper, each containing a single datapoint.

In many programming languages, including Java and Python, data composites can take the form of *objects*, which contain both the data and functions for operating on their own data. The object's functions use special syntax to access that object's data, such as the `self` reference in Python. An object can also provide different visibility rules that specify whether its internal data is publicly accessible outside the object's own functions or only privately accessible.

In an attempt to be general, we will treat composite data structures in their most general form: as a collection of data. While example code snippets in this book and elsewhere may implement the composite data structures as objects, the algorithms can be adapted to use non-object representations as well. In code that uses composite data structures or objects, we use the syntax of `composite.field` to indicate accessing a particular field of a composite data structure. For example, using the following:

```
latest_record.name = "Sublime Blend"
```

we set the `name` field of the record `latest_record` in our coffee log to have the value `Sublime Blend`.

Arrays

An *array* is generally used to store multiple related values. For example, we might want to track the amount of coffee consumed daily over a year. We could brute-force the storage by creating 365 individual variables, such as `AmountDay1`, `AmountDay2`, `AmountDay3`, and so forth, but this is tedious to type and doesn't allow us to use any structure for the data. `AmountDay2` is only a textual tag, so the program doesn't know that `AmountDay1` stores information for the day before and `AmountDay3` for the day after; only the programmer knows this.

Arrays provide a simple mechanism for storing multiple values in adjacent and indexable bins. An array is effectively a row of variables—a contiguous block of equal-sized bins in the computer's memory, as in *Figure 1-3*. Like individual vari-

ables, arrays occupy a chunk of memory and can sit adjacent to arbitrary other information. Each of the array's bins can store a value of the given type, such as a number, a character, pointer, or even other (fixed-size) data structures.

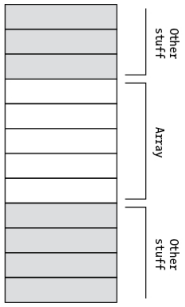


Figure 1-3: Arrays as bins in the computer's memory

Arrays appear throughout our real-world daily lives as well. The row of lockers lining a high school hallway, for example, is a physical array for storing students' coats and books. We can access any individual storage container by just opening the corresponding locker.

The structure of an array allows you to access any value, also known as an *element*, within the array by specifying its location, or *index*. The bins occupy adjacent locations in the computer's memory, so we can access individual bins by computing their offset from the first element and reading the memory in that location. This requires just a single addition and memory lookup regardless of which bin we access. This structure makes arrays especially convenient for storing items that have an ordered relationship, such as our daily coffee intake tracker.

Formally, we reference the value at index `i` of array `A` as `A[i]`. In our locker example, the index would be the number displayed on the front of the locker. Most programming languages use *zero-indexed arrays*, which means the first value of the array resides at index 0, the second at index 1, and so forth, as shown in *Figure 1-4*.

Value:	3	14	1	5	9	26	5	3	5
Index:	0	1	2	3	4	5	6	7	8

Figure 1-4: A zero-indexed array

We will use zero-indexed arrays throughout this book, to stick to general computing convention. *Figure 1-5* represents how zero-indexed arrays appear in the computer's memory, where the white spaces are the elements of the array.

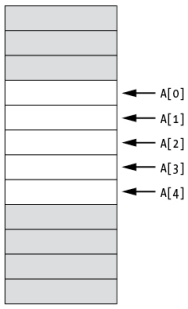


Figure 1-5: A zero-indexed array arranged in computer memory

Zero-indexing conveniently allows us to compute an element's location in memory as an offset from where the array starts in memory. The location of the *i*th item in the array can be computed by:

Location(*item* i) = *Location*(*start of array*) + *Size of each element* × i

The location of the element at index zero is the start of the array. For example, the fifth element of the example array `A` in *Figure 1-5* would be `A[4]`, and, going by the values indexed in *Figure 1-4*, contain the value 9.

NOTE

It's possible to start an index at 1, too, and some programming languages do use that convention. The equation for a bin's address in a one-indexed array would be Location(start of array) + Size of each element × (i –1).

In most programming languages, we get and set values in an array using a combination of the array's name and the index. For example, we might set the value of the bin with index 5 equal to 16:

```
A[5] = 16
```

For our coffee-tracking example, we could define an array `Amount` to store the number of cups consumed in a day and store the corresponding counts in `Amount[0]` through `Amount[364]`. The single array allows us to access 365 distinct values, in order, through a single variable name. We have transitioned from a series of similarly named, but independent, variables to a mathematical offset of a single location. To understand the power of this, consider our school lockers. Naming individual lockers "Jeremy's Locker" or "Locker for the third student with a last name starting with K" would make them nearly impossible to find quickly. Rather than simply accessing a specific index, students would have to check a large number of lockers, comparing textual tags until they found the correct match. With array indexing, the students can just use its offset to determine where the locker is and access it directly.

Although we often visualize and discuss arrays as the whole data structure, it is important to remember that each bin behaves like an individual variable. When we want to make a global change to the array, such as shifting the elements forward one position, we need to apply the change individually to each bin as shown in *Figure 1-6*.

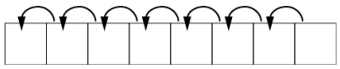


Figure 1-6: Shifting elements in an array forward, bin by bin

Arrays aren't like books on a bookshelf. We can't shove the entire collection over at once to make room for the newest edition of *Coffee Lover's Guide to the Best Free-Trade Coffees*. Arrays are more like a row of storefronts. We can't just squeeze a new coffee shop between our favorite neighborhood bookseller and barbershop. To make space, we'd need to shift the storefronts down one by one, by emptying each store and moving its contents into the adjacent building.

In fact, we have to juggle values simply to swap two values in an array. To swap the values at some indices `i` and `j`, for instance, we need to first assign one of them to a temporary variable:

```
Temp = A[i]
A[i] = A[j]
A[j] = Temp
```

Otherwise, we would overwrite the value in one of the bins and the two bins would end up with the same value. Similarly, if we are swapping the locations of the coffee shop and bookseller, we first need to move the contents of the bookstore into an empty third location in order to make space for the contents of the coffee shop. Only after we've moved the coffee shop can we move the bookstore's contents from the temporary third location into the coffee shop's former location.

Insertion Sort

The best way to understand the impact an array's structure has on how it can be used is to examine it in the context of an actual algorithm. *Insertion sort* is an algorithm to sort the values in an array. It works on any type of value that can be ordered. We could sort integers, strings, or even the coffees in our pantry by expiration date.

Insertion sort works by sorting a subset of the array and expanding this sorted range until the entire array is in order. The algorithm iterates through each element in the unsorted array and moves it down into the correct location of the sorted section. At the start of iteration `i`, the items in bins 0 through `i` – 1 are all in sorted order. The algorithm then takes the item at index `i`, finds the correct location in the sorted prefix, and inserts it, shifting the necessary items down to make room. The sorted prefix has now grown by one—bins 0 through `i` are in sorted order. We can start at `i` = 1 by declaring the first element to be our initial sorted prefix.

Say we want to sort our coffee collection in order of freshness—after all, it would be tragic to leave a bag of premium coffee languishing at the back of the pantry until it was stale. We need to move the earliest best-by dates to the left side of the shelf where they can be readily accessible.

We begin our coffee insertion sort by proclaiming a single bag at the front to be *sorted* and using this range as our sorted prefix. We then look at the second bag on the shelf and compare dates to determine whether it should go before the first one. After we swap the order, or determine that a swap isn't necessary, we can confidently proclaim that the first two elements are sorted. We have a subset that's fully sorted. We then progress to the third bag and determine where it should sit relative to the first two, perhaps making a few swaps in the process. This process continues down the shelf until we've achieved perfect coffee organization.

We can implement insertion sort with a pair of nested loops, as shown in *Listing 1-1*.

```
InsertionSort(array: A):
    Integer: N = length(A)
    Integer: i = 1
  ❶ WHILE i < N:
        Type: current = A[i]
        Integer: j = i - 1
      ❷ WHILE j >= 0 AND A[j] > current:
            A[j + 1] = A[j]
            j = j - 1
```

```
            A[j + 1] = current
            i = i + 1
```

Listing 1-1: Implementing insertion sort with nested loops

The outer loop with iterator `i` starts at the first unsorted element, `i = 1` and progresses through each value in the unsorted range ❶. The inner loop with iterator `j` shifts the current value down into the sorted prefix ❷. At each step, we check the position within the sorted prefix by comparing the `current` value to the preceding location in the prefix, index `j`. If the element at `j` is larger, the two values are in the wrong order and must be swapped. Since we are storing the current value in a separate variable `current`, we copy the data from the preceding bin directly. There is no need to do a full swap. The inner loop continues until it shifts the current value to the front of the array or it finds a preceding value that is smaller, which indicates the current value is in the correct location of the sorted prefix. We only need to write the current value at the end of the loop when it is in the correct location. The outer loop then proceeds to the next unsorted value.

We can visualize the behavior of the algorithm as shown in *Figure 1-7*. Each row shows the state of the array at the beginning of the iteration. The shaded box represents the current item being shifted into position, and the arrows represent the corresponding shifts.

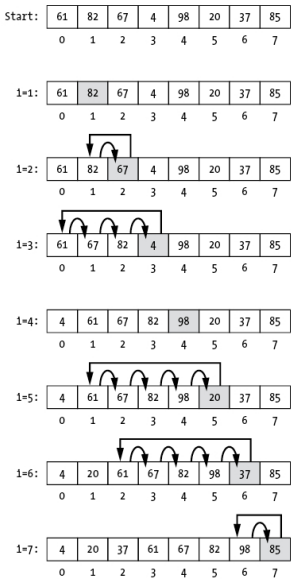


Figure 1-7: Visualization of an insertion sort algorithm

Insertion sort isn't particularly efficient. When inserting elements into the array, we could end up shifting around significant portions of the array. In the worst case, the cost of the algorithm scales proportionally with the square of the number of items—for every item in the list, we shift all the items in front of it. If we double the size of the array, we increase the worst-case cost by a factor of four. While this may not be a huge cost in our coffee pantry, where we are likely to keep only a small number of coffees that we can consume before they go stale, the quadratic cost of the algorithm skyrockets in many applications.

Yet insertion sort provides an important insight into how arrays function. Within this simple algorithm we illustrate several attributes of the array, including the power of being able to access items by their index, the ability to swap values when inserting new elements, and the valuable ability to iterate over entries.

Strings

Strings are ordered lists of characters that can often be thought of as a special kind of arrays. Each bin in the string holds a single character, be that a letter, number, symbol, space, or one of a limited set of special indicators. A special character is often used to indicate the end of the string, as represented by the `/` in the last bin in *Figure 1-8*. Characters in strings can often be accessed directly using their index.

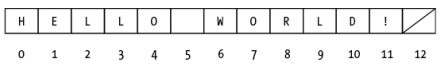


Figure 1-8: A string spelling "Hello world!"

In some programming languages, strings are directly implemented as simple arrays of characters. In others, strings may be objects, and the string class serves as a wrapper around an array or other data structure holding the characters. The wrapper class for a string provides additional functionality, such as the ability to dynamically resize the string or search for a substring. In either case, it is useful to think about how the general array-like structure impacts operations on the string. When we display a string on the computer screen, we are effectively iterating through each of its characters and displaying them one at a time.

The common test of equality is more interesting to consider. Unlike integers, which can be directly compared with a single operation, strings must be com-

pared by iterating through each character. The program compares it individually to its counterpart and returns whether it finds a mismatch.

Listing 1-2 shows the algorithm for checking the equality of two strings. The algorithm starts by comparing the strings' size. If they are not the same length, the algorithm stops there. If they are the same length, the algorithm iterates through each position and compares the respective letters of each string. We can terminate the loop as soon as we find a single mismatch. Only if we make it all the way to the end of the strings without a mismatch can we declare the strings equal.

```
StringEqual(String: str1, String: str2):
    IF length(str1) != length(str2):
        return False
    Integer: N = length(str1)
    Integer: i = 0
    WHILE i < N AND str1[i] == str2[i]:
        i = i + 1
    return i == N
```

Listing 1-2: The algorithm for checking the equality of two strings

Figure 1-9 demonstrates how this algorithm operates on two strings. The equality sign indicates which pairs of characters matched when compared. The X represents the first mismatched pair, where the test terminates.

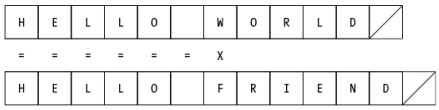


Figure 1-9: A comparison of two strings

The worst-case computational cost of string comparison grows proportionally with the length of the strings. While the work required to compare two small strings can be negligible, the same operation on two long strings can be time-consuming. For comparison, imagine the tedium of scanning through two editions of the same book, letter by letter, looking for each difference in the arrangement of text from one book to the next. In the best case, we find a mismatch early. In the worst case, we need to examine the majority of the book.

Many programming languages, such as Python, provide a string class that allows direct comparisons, so we never need to implement the comparison code in Listing 1-2 directly. Still, underneath the simple comparison function lies a loop that iterates over all the letters. Without understanding this vital detail, it is possible to vastly underestimate the cost of string comparisons.

Why This Matters

Variables and arrays are staples of introductory programming classes and thus might seem less than exciting, but they are important to examine because they provide the very foundations for computer programming and data structures. These concepts also provide the baseline against which to evaluate dynamic data structures and their impact on algorithms. In later chapters, we will see how dynamic data structures can offer different tradeoffs among efficiency, flexibility, and complexity.

2
BINARY SEARCH

Binary search is an algorithm for efficiently searching a sorted list. It checks the sorted list for a target value by repeatedly dividing the list in half, determining which of the two halves could contain the target value, and discarding the other half. This algorithm's simplicity of logic and implementation make it a perfect introductory topic for computer science, so binary search algorithms are nearly universal throughout computer science courses and textbooks.

The skeptical reader might wonder, "How often will I really need to search a sorted list?" or, more accurately, "How often will I need to implement a function to search my sorted list? Haven't a few million people done this? Isn't it in a library somewhere?" While you shouldn't reject the possibility of one day needing to implement your own binary search, its true importance goes well beyond its implementation.

Binary search illustrates how clever algorithms can use the structure in which data is stored to achieve significant computational savings, even when this structure is as simple as sorted data. Binary search is easily analyzed for correctness and efficiency, provides guarantees of both speed and correctness, and demonstrates the fundamental interaction between data and algorithms. It's an excellent lens with which to examine difference in data storage techniques, such as the difference between linked lists and arrays, or the motivation behind many tree-based algorithms. It can even be used to create a better cup of coffee.

The Problem

Before defining any new algorithm, we must define the problem the algorithm will try to solve. In this chapter, our aim is to find a single item in a list that matches the given target value; we need an algorithm that can efficiently perform such a search. Formally we can define this search as:

> *Given a set of N data points $X = \{x_1, x_2, \ldots, x_N\}$ and a target value x', find a point $x_i \in X$ such that $x' = x_i$, or indicate that no such point exists.*

In our everyday lives, we would likely describe the task as "Find me this particular thing." This search problem is one that we all face numerous times a day. We might be hunting for a word in a dictionary, a name in our contacts list, a specific date in a list of historical of events, or our preferred brand of coffee in a densely packed supermarket shelf. All we need is a list of candidates and a way to check whether we've found a match.

Linear Scan

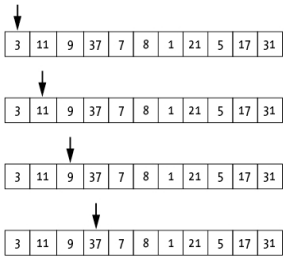
To appreciate the advantages of binary search, we start with a simpler algorithm, *linear scan*, to provide a baseline for comparison. Linear scan searches for a target value by testing each value in our list, one after the other, against the target value, until the target is found or we reach the end of our list. This is how the author normally searches supermarket shelves—running his finger along the brightly colored packages of coffee, one by one, while mumbling to himself about the need for better indexing schemes.

Imagine that we are looking to find the target value in an array `A` of numbers. In this case, let's use `target = 21`. We iterate through each bin in the array and check whether that value equals 21, as illustrated in *Figure 2-1*.

Listing 2-1 shows the code for linear scan. The code returns the index of the matching element if one is found and returns an index of `-1` if the search fails and the item is not in our array.

```
LinearScan(Array: A, Integer: target):
    Integer: i = 0
    WHILE i < length(A):
        IF A[i] == target:
            return i
        i = i + 1
    return -1
```

Listing 2-1: The code for linear scan



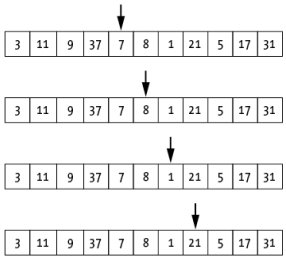


Figure 2-1: A linear scan over an array of integers

A single `WHILE` loop iterates over each element of the array, and the internal `IF` statement compares that element to the target. As soon as we come across an element matching the target, we return the corresponding index. If we make it to the end of the array, we return `-1`.

Linear scan isn't fancy or clever. It's a *brute-force* test guaranteed to find the item of interest (if the item is in the data) because it checks every possible item until it finds a match or confirms the item is missing. This is thorough but inefficient, especially for large lists. If we know nothing about the structure of the data in `A`, there is nothing we can do to streamline the process. The target value could be in any bin, so we may need to check them all.

To illustrate linear scan's limitations, imagine performing such a scan on a physical row of items, like a line of introductory computer science students standing outside a classroom. The teacher, looking to return a specific student's assignment, walks down the line asking each student "Is your name Jeremy?" before potentially moving onto the next. The search stops when the teacher finds the correct student or makes it to the end of the line. The students (correctly) roll their eyes and mutter about their inefficient instructor.

Sometimes there are ways to make linear search faster *per comparison*. For example, we might optimize the comparison time for more complex data by stopping at the first mismatched letter when comparing strings, as described in Chapter 1. Likewise, in the supermarket case, we could consume massive amounts of coffee ahead of time, so our shaky finger zips faster along the shelf. However, this only helps to a point. We are still limited to checking every item, one by one.

In the next section, we'll see how a small amount of structure in the data changes everything.

Binary Search Algorithm

Binary search is an algorithm to find a target value v in a *sorted* list and only works on sorted data. The algorithm can be written to work with data sorted in either increasing or decreasing order, but, for now, let's consider the case of data sorted in increasing order—lowest to highest. The algorithm operates by partitioning the list in half and determining in which half v must reside. It then discards the half that v is not in and repeats the process with only the half that can possibly still contain v until only one value remains. For example, if we were searching the sorted list in *Figure 2-2* for the value 7, we would find 5 at the midpoint and could rule out the first half of the list. Anything before the middle element cannot be greater than 5, and, since 5 is less than 7, everything before 5 is also less than 7.

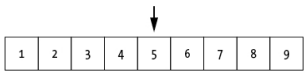


Figure 2-2: A sorted list of integers from 1 to 9, where 5 is the midpoint

The key to efficient algorithms is using information or structure within the data. In the case of binary search, we use the fact that the array is sorted in increasing order. More formally, consider a sorted array `A`:

$A[i] \leq A[j]$ for any pair of indexes i and j such that $i < j$

While this might not seem like a lot of information, it's enough to allow us to rule out entire sections of the array. It's similar to the logic we use to avoid the ice cream aisle when searching for coffee. Once we know an item won't be in a given area, we can rule out that entire set of items in that area without individually checking them.

Binary search tracks the current search space with two bounds: the upper bound `IndexHigh` marks the highest index of the array that is part of the active search space, and the lower bound `IndexLow` marks the lowest. Throughout the algorithm, if the target value is in the array, we guarantee the following:

```
A[IndexLow] ≤ v ≤ A[IndexHigh]
```

Binary search starts each iteration by choosing the midpoint of the current search space:

```
IndexMid = Floor((IndexHigh + IndexLow) / 2)
```

where `Floor` is a mathematical function that rounds a number down to an integer. We then compare the value at the middle location, `A[IndexMid]`, with the target value `v`. If the middle point is less than the target value, `A[IndexMid] <`

`v`, we know the target value must lie after the middle index. This allows us to chop the search space in half by making `IndexLow = IndexMid + 1`. Alternately, if the middle point is greater than the target value, `A[IndexMid] > v`, we know the target value must lie before the middle index, which allows us to chop the search space in half by making `IndexHigh = IndexMid – 1`. Of course, if we find `A[IndexMid] == v`, we immediately conclude the search: we've found the target. Boisterous celebration is optional.

Each row in *Figure 2-3* represents a step in the binary search process on a sorted array. We're searching the array at row (a) for the value `15`. At the start, our search bounds include the entire array: `IndexLow = 0` and `IndexHigh = 11`.

In row (b), we compute the midpoint (rounding down) to be `IndexMid = 5`. Comparing the midpoint's value to the target value, we see that `A[5] = 11`, which is less than our target value of 15. Therefore, in row (c), we rule out every element in the array up to and including index `5`—that is, all the shaded elements—by adjusting the lower bounds: `IndexLow = 6`. We've eliminated almost half our search space with a single comparison! The algorithm repeats this process on the remaining range, computing the new midpoint as `IndexMid = 8`, comparing to the target value (`A[8] = 30`, which is greater than `v = 15`), and refining our bounds to `IndexHigh = 7`. In row (d), we once again eliminate half of the remaining search in the same way. In row (e), we again compute the midpoint as `IndexMid = 6` and compare it to the target value (`A[6] == v`). We've found the target!

Note that even though the lower bound's index pointed to the target value (`v = 15`) for several iterations, we continued the search until the *midpoint* pointed to the target value. This is because our search checks only the value at the midpoint against the target and not the values at the lower or upper indexes.

Returning to our line of introductory computer science students, we can imagine that by the end of the semester, the teacher asks the students to line up in alphabetical order. The teacher then starts a binary search by asking the middle student "What is your name?" and uses the responses to prune out half the line. The professor then mentally revises the bounds, moves to the new midpoint, and repeats the process. Thus, the professor can turn the exercise of returning an assignment into a demonstration of binary search—while also covering up the fact that they never managed to learn the students' names.

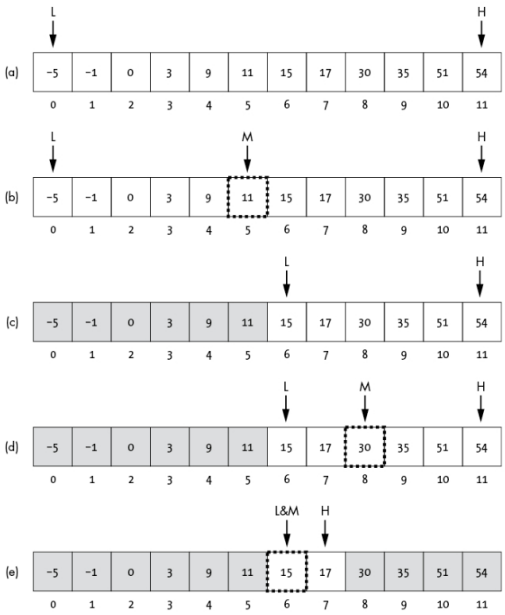
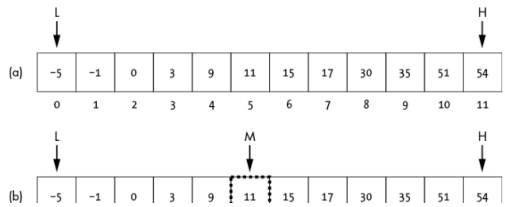


Figure 2-3: A binary search for the value 15 over a sorted array

Absent Values

Next, we need to consider what happens if the target value is not in the list and how the binary search confirms the value's absence. In the linear scan case, we know that an element is not in the list as soon as we hit the end of the list. For binary search, we can conclude that our target item does not exist by testing the bounds themselves. As the search progresses, the upper and lower bounds move closer and closer until there are no unexplored values between them. Since we are always moving one of the bounds *past* the midpoint index, we can stop the search when `IndexHigh < IndexLow`. At that point, we can guarantee the target value is not in the list. *Figure 2-4* shows an example search for `v = 10` on a sorted array, where `10` does not appear in the array.



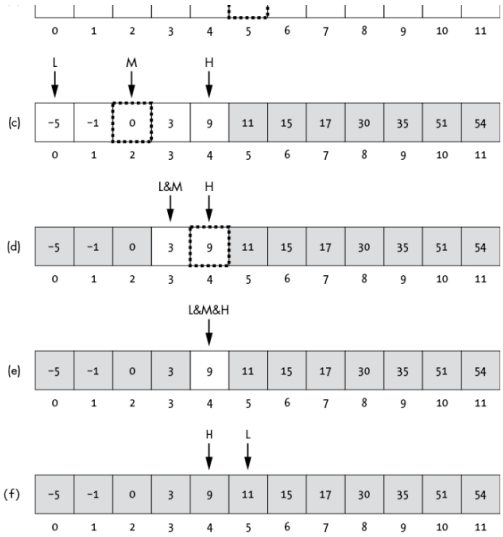


Figure 2-4: A binary search for a value (10) that isn't in the array

We could theoretically have stopped the search earlier than row (f): once the value at our high bound is less than the target value (`IndexHigh = 4`) we know that the target can't be in the array. However, as with our search in *Figure 2-3*, the algorithm only checks the value at the midpoint against the target. It tracks the indices of the high and low bounds, but does not check the values at these locations explicitly. While we could add logic to capture this case, as well as the corresponding case of the lower bound being greater than the target value, we will keep the logic simple for now.

Implementing Binary Search

We can implement a binary search in code with a single `WHILE` loop, as shown in *Listing 2-2*. Like the code for linear search in *Listing 2-1*, the binary search algorithm returns the index of the target element if it is in the array. If there is no matching element in the array, the algorithm returns `-1`.

```
BinarySearch(Array: A, Integer: target):
    Integer: IndexHigh = length(A) - 1
    Integer: IndexLow = 0
 ❶ WHILE IndexLow <= IndexHigh:
     ❷ Integer: IndexMid = Floor((IndexHigh+IndexLow) / 2)

       IF A[IndexMid] == target:
           return IndexMid
       IF A[IndexMid] < target:
         ❸ IndexLow = IndexMid + 1
       ELSE:
         ❹ IndexHigh = IndexMid - 1
    return -1
```

Listing 2-2: Implementing binary search with a single loop

While the high and low indices have not crossed, we continue the search ❶. During each iteration, we compute a new midpoint ❷ and check the midpoint value against the target. If it's an exact match, we've found our target and can directly return the corresponding index. If the value at the midpoint is too small, we adjust the lower bounds ❸. If the value is too high, we adjust the upper bounds ❹. If `IndexHigh < IndexLow`, the element is not in the array, so we return `-1`.

Depending on the programming language, we could use approaches other than returning `-1` to indicate failure, such as throwing an exception. Regardless of the actual mechanism, your code and documentation should always be absolutely clear about what happens if the element is not in the array so that callers of the function can use it correctly.

Adapting Binary Search

So far, we have considered binary search in the context of lists and arrays—fixed sets of discrete items. It is easy to see how we could bring this algorithm into the real world by applying it to a shelf of sorted books, names in a telephone book, or a clothing rack ordered by size. But we can adapt this same approach to continuous data, where we don't start with a set of individual items or indices, as well. Instead, we use high and low bounds on the values themselves.

Imagine you aim to craft the perfect cup of coffee. After months of laborious research, you've confirmed the optimal temperature and quantity of water. Yet one mystery remains: What quantity of coffee grounds should you use? Here the sources diverge. The Strong Coffee camp recommends a tremendous 5 tablespoons of coffee grounds, while the Watery Coffee camp recommends a paltry 0.5 tablespoons.

The problem of determining your own optimal scoop of coffee grounds lends itself perfectly to a binary search, as shown in *Figure 2-5*. We start with reasonable upper and lower bounds as illustrated in *Figure 2-5*(a).

LowerBound = 0 tablespoons The "coffee" was a cup of warm water.

UpperBound = 5 tablespoons The coffee was too strong.

The true value must be somewhere in between. Note that our bounds are now the values themselves instead of item indices.

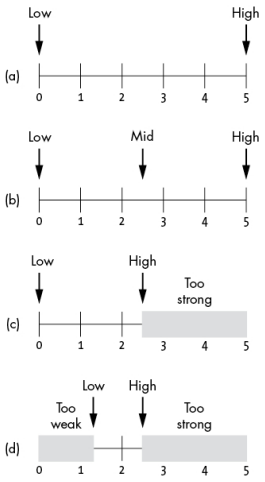


Figure 2-5: An adapted binary search can be used to search a range of real numbers.

As with a binary search on an array of values, we can define the midpoint at 2.5 tablespoons and test that (*Figure 2-5*(b)). Again, 2.5 tablespoons is just a value. It doesn't correspond to an element in an array or item on the shelf. We do not have an array of predetermined values, but rather the infinite range of all real numbers between 0.0 and 5.0, and any individual measurement effectively corresponds to an index into that range.

We find the coffee made from 2.5 tablespoons is a little too strong for our taste, allowing us to refine the bounds. Our optimal amount of coffee is now constrained to lie between 0 tablespoons and 2.5 tablespoons (c). Our search continues with a new midpoint of 1.25 tablespoons, an amount that produces a weak brew. We need to refine our lower bound (d).

The search for morning bliss continues this way until we've sufficiently narrowed down our range. Unlike with a discrete array of values, we may never find the exact point that satisfies our search. After all, there is an infinite number of real values. If our optimal amount of coffee is 2.0 tablespoons, we might try values of 2.50, 1.25, 1.875, 2.1875, and 2.03125 before concluding that we are close enough. Therefore, we terminate the search when our range is sufficiently small:

```
UpperBound – LowerBound < threshold
```

Contrast this search with a linear scan through the options. In the name of science, we may resolve to try every possible increment of 0.05 tablespoons until we find the optimal brew. After all, this is coffee, and we must be thorough. Starting at our low index (0.0 tablespoons—alternately known as a cup of warm water), we continually increment the amount by 0.05 and retest. We run through 0.05, 0.10, 0.15, . . . , 1.00 before we start to get to a reasonable strength. We would need many trials to get to the correct point, at least 20 of which would be too weak to even to count as coffee. That's a lot of wasted effort and coffee beans.

The use of binary search also allows for better precision. By sampling only 0.05 increments during our linear scan, we are limited to how close to the target value we can get. Binary search keeps homing in on a smaller range until we stop. We choose what value of `UpperBound – LowerBound` is sufficient to halt the search, allowing us to narrow it down to within 0.0001 tablespoons or closer.

This adaptation of the binary search approach forms the basis of important mathematical techniques, such as bisection search. *Bisection search* uses it to find the zero of a function, or the value of x such that $f(x) = 0$. Instead of evaluating whether coffee is too strong or too weak, bisection search tracks bounds where the function is above zero and below zero. By repeatedly dividing the interval in half at the midpoint, the algorithm zooms in on the value of x where the function is exactly zero.

Runtime

Intuitively, we can see that binary search is often faster than a linear scan of the data. Let's find out how much faster binary search can be to determine whether it's worth the additional code complexity.

Of course, the relative speed of the two algorithms depends on the data itself. If we are searching for values that always occur at the start of the list, linear scan will win. Similarly, binary search might be unnecessary for tiny lists. We don't need to partition a list in half if it has only two elements. We can just look at those elements directly.

We often analyze the runtime of an algorithm in terms of its average and worst-case performance as the size of the data N grows. Computer scientists often use measures such as Big-O notation to more formally capture those concepts. We won't formally analyze algorithms in this book or use Big-O notation, but we will consider the same two aspects throughout for each algorithm:

- The average-case runtime of an algorithm as the size of the data grows
- The worst-case runtime of an algorithm as the size of the data grows

For now, let's compare worst-case performance for linear scan and binary search. For linear scan, the worst case occurs when the target value is at the end of the list or not in the list at all. In these cases, the algorithm has to check every single value. If the array has N values, it will require N comparisons. Its worst-case running time is *linear* with the size of the data.

In contrast, even the worst-case binary search will discard half the data at each step, so the number of comparisons is *logarithmic* with the size of the data set. It scales proportional to $\log_2 N$, which is the base-2 logarithm of N. Admittedly there is more work per step: instead of checking a single value, we have to move our bounds and compute a new midpoint. However, for large enough lists, the benefit of needing only a logarithmic number of comparisons will far outweigh additional per-step costs.

Why This Matters

This fixation on binary search in introductory computer science classes isn't the result of binary-search advocacy campaigns, fan clubs, or secret societies (although those would all be understandable). Rather, it's binary search's simplicity that makes it a perfect introductory topic. It is a clean and effective example of one of the most fundamental concepts of computational thinking: that designing algorithms by using the structure in the problems themselves helps us construct efficient solutions. By taking advantage of the sorted nature of the data, we are able to cut the worst-case runtime from linear with the number of values to logarithmic—a difference that becomes more significant as the data grows.

Throughout the rest of the book, we will continue to look at the tight relationship between problem structure (including within the data) and how we can create efficient solutions.