Data Structures

Andrew Rosen

Contents

Part I

Preliminaries

Chapter 1

Introduction

1.1 What is a Data Structures Course

Data Structures is all about defining the different ways we can organize data. This is not databases, which is concerned with defining the various attributes of a bunch of data; this is much more granular. We want to know how to store and retrieve a single item of data.

1.2 Why This Book?

This textbook is free.

It is both Java and Python, which is a bit insane. You have two valid choices:

- Understand that the concepts we are learning are way more important than the language and treat the other language as psuedocode (which isn't hard for Python)
- Be comfortable in multiple languages and embrace being a polyglot. Impress your friends, wow your rivals!

1.2.1 Where Does This Book Fit Into a Computer Science Curriculum

Education in Computer Science is based around three core topics: translating the steps of solving a problem into a language a computer can understand, organizing data for solving problems, and techniques that can be used to solve problems. These courses typically covered in a university's introductory course, data structures course, and algorithms course respectively, although different universities decide exactly what content fits in which course. Of course, there is are lot more concepts in computer science, from operating systems and low level programming, to networks and how computers talk to each other. However, all these concepts rely on the knowledge gained in the core courses of programming, data structures, and algorithms.

This textbook is all about Data Structures, the middle section between learning how to program and the more advanced problem solving concepts we learn

in Computer Science. Here, we focus on mastering the different ways to organize data, recognize the internal and performative differences between each structure, and learn to recognize the best (if there is one) for a given situation.

1.2.2 What Are My Base Assumptions about the Reader?

This textbook assumes that the student has taken a programming course that has covered the basics. Namely: data types such as ints, doubles, booleans, and strings; if statements, for and while loops; and object orient programming. This book is also suitable for the self taught programmer who has not learned much theoretical programming

1.3 To The Instructor

1.3.1 Professor Rosen's Extremely Opinionated Advice on How to Lecture

You'll note that this textbook lacks some of the features found in commercially available textbooks. The biggest of these is slides. For the most part, slides are too static to help students understand how to code.

I'll go a step further to be blunt: from intro to programming all the way thru data structures, slides are absolute trash; use them if and only if you have no time to prepare. In fact, even if you have no time to prepare, I would caution against using it.

I have been teaching Data Structures since Fall 2011. In order of preference, this is how I would tackle *this* class, which I fully recognize may not work for you or your teaching style.

Lecture with slides. Do this if slides are available (they are not for this book) and it is your first time teaching this class.

Lecture via live coding. Basically, your lecture unit for a data structure should look like this[1]:

- Introduce the ADT that you will be modeling. So for a [array]list, describe what it is, why we want it over an array, and the operations.
- Code a functional, pedagogical implementation live in class.
  - Functional means a student could ostensibly use it in an assignment. This means most of the work will focus on `add` and `remove` or their equivalents.
  - Pedagogical means that you should keep straightforward and not try to reproduce the entire built-in class. If you're working in Java, your implemented `MyArrayList` shouldn't try to implement the `List` interface. You should only focus on the primary ways a programmer interacts with the class in question. It also means you should emphasize that the built-in, real-world classes will have a number of optimizations that speed things up , but what you're covering is a close enough approximation.

[1] Conviently, the textbook is written for you to model this

- This might be a bit unnerving to have to reproduce a class in front of the class, but watching someone program these things from scratch works better than just reading snippets of text.
- Mistakes will be made, but students need to see mistakes are normal.

Do the above, but flip the lecture. You can see my example of this here:

1.3.2 Exercises

Does the lack of varied exercises make cheating on assignments easier as semesters go on? Yes, but that bridge was burned long ago. The cheating student can plagiarize from various websites or anonymously hire another to do their work for them. However, the student who cheats isn't exactly clever and certainly hasn't been exposed to much game theory. They will often cheat from the same source.

In addition, during the writing of this text, technologies such as GPTChat were released. This hasn't so much burned the bridge as dropped napalm on the entire surrounding forest. Newer technologies will then salt that earth. I recommend an open and honest dialogue with your students and at least 50% of their grade being the result of evaluations and assessments you do in class. This can range from proctored exams to flipping the classroom and giving students the chance to work on homework in class, where they are much more likely to turn to you or their peers for help.

My personal solution for assignments is to use require students to demo their homework to me or a TA to receive a grade. As part of this demo, they must answer questions about their solutions. Now, as you well know, a student being unable to answer question about their work or make on the fly adjustments **isn't necessarily** an indicator that a student has cheated. Personally, I find half of my students seem to lose approximate 50 IQ points upon being directly questioned. It is daunting to be the sole subject of attention to the person who is making the determination as to whether you pass or fail. But I digress. To summarize, being unable to answer questions about their code might be an indication of cheating, it might be an indication of nerves. In most cases, you can figure out if it's the latter case, in which case, just send them away until they can explain their code. Don't penalize them, but remind them that programming interviews require this kind of presentation of skills.

As far as student who use AI, AI generated code has a number of tells and those tells will change over time. However, the usual marker is using either too well documented code and data structures or syntax well above what you would expect from a student. Now, if the student is using Vim or Emacs or rocking Linux, they probably can explain exactly what they are doing. In fact, save yourself the trouble and just assign them a minimum of a B. However, most of the time a student won't be able to explain the thought process behind the solution or the way some syntax works. If pressed they will explain a vague "friend" helped them with the code. You can press further if you want and handle it however. Personally, it depends on where we are in the semester. In the first few weeks, I emphasize that they will be completely wrecked by the exams if they rely on this "friend" and they need to do the work themselves and tell them to come back when that is the case. At the end of the semester, I am much less inclined to have mercy.

1.3.3 The Order

1.3.4 Assignments

I will drop the sporadic assignment here or there, drawing from the same places you should draw from:

- Nifty Assignments
- Problem Solving with Algorithms and Data Structures using Java (Miller)

1.3.5 How to Use

1.4 To The Student

Why are we learning this? As Brad Miller and David Ranum put in their aforementioned book (which is creative commons and you should totally check out):

> To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. These models allow us to describe the data that our algorithms will manipulate in a much more consistent way with respect to the problem itself.
>
> Earlier, we referred to procedural abstraction as a process that hides the details of a particular function to allow the user or client to view it at a very high level. We now turn our attention to a similar idea, that of data abstraction. An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user' s view. This is called information hiding.
>
> Figure 2 shows a picture of what an abstract data type is and how it operates. The user interacts with the interface, using the operations that have been specified by the abstract data type. The abstract data type is the shell that the user interacts with. The implementation is hidden one level deeper. The user is not concerned with the details of the implementation.
>
> Figure 2: Abstract Data Type
>
> The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types. As we discussed earlier, the separation of these two perspectives will allow us to define the complex data models for our problems without giving any indication as to the details of how

the model will actually be built. This provides an implementation-independent view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.

1.4.1 How to use

1.5 License

This work is funded by Temple University's North Broad Press and is under Creative Commons - Attribution Non Commericial License

1.6 On Styles

On styles: Java convention is to use camel case for variable types (`myVariableName`), while python convention is to use underscores (`my_variable_name`). I will be using the Java style camel-casing for variables throughout the book for consistency and because it is my preference.

Chapter 2

Functions and How They Work

This will be an extremely short chapter, but an important one. We are already going to assume that you know what a function, a method, or procedure is and that you have written them before. After all, Data Structures is a point continuing your education in programming, not beginning it. That said it is possible that you missed some subtleties along the way.

That's understandable - programming is a very large topic and there's more than enough concepts that no one who graduates with a degree in computer science can be expected to be an expert in every area any more.

With this in mind, let me take the time to review some subtleties surrounding the vocabulary of functions.

2.1 Function vs Method

You'll often hear programmers use these two terms interchangeably to refer to what essentially amounts to a subprogram. But what is the difference? I like to explain it this way: functions are the verbs of the programming language. When we create a new function, we a creating a new verb in the programming language we are working in. Methods are a special type of function that are closely linked to objects; they are the actions or verbs you want your objects to perform.

Java blurs this a bit with `static` methods, but for the purposes of this text, when I write *method*, I am talking about Java's *instance methods*. *Function*, in the context of this book, is analogous to *static methods*. Similarly, if you are coming from Python, when I say function, I am talking about a boring top-level, unindented function such as the ones you've been writing since you first learned Python. Method would refer to the functions you create as part of your classes.

2.2 Argument vs Parameter

An argument is the actual value you pass in, the parameter is the variable that accepts it.

Listing (Java) 2.1: Java Parameter vs Argument Example

```java
public static void doubleThis(int num) {   // parameter
    return 2*num;
}

public static void main(String[] args){
    int x = 7;
    int y = doubleThis(x);  // argument
}
```

Listing (Python) 2.2: Python Parameter vs Argument Example

```python
def doubleThis(num): # parameter
    return 2*num

x = 7
y = doubleThis(x) # argument
```

In the above examples, `x` is an argument and `num` is a parameter.

2.2.1 Does anyone actually care?

I cared enough to look it up, but I also had to look it up to double check that I'm correct and I keep coming back to this page as a reference for myself. In a casual situation or talking with another programmer, everyone will be able to *grok* your meaning from the context, as you just did with the word `grok`. I would take care to get it correct for your assignments and exams, much like you would take care to avoid using "ain't" in a formal essay. One professor might be a stickler about it and one might not care.

2.3 Passing Arguments

The vast majority of programming languages are *pass by copy* with a huge honking asterisk.

- Pass by copy means that when something is input as the argument to a function (or method), the function gets a copy of the thing you are passing to it.
- The *huge honking asterisk* is that you are almost always passing a *reference* or *pointer* to an object, not the object itself. The reason for this is that if we had a super mega huge object, copying it would take up a super mega huge amount of time and memory.

2.3.1 How it Works in Java

In Java, we have two broad categories of data types: primitives and objects.

When you pass a primitive, such as an `int` or `double`, the value gets copied from where it is stored in memory and copied into the argument.

When you create an object, such as with `Scanner scan = new Scanner(System.in);`, the variable `scan` will hold not the Object that was created by the constructor, but the *memory location*, or *reference* of where to find it. Look at the code below where we are we create an array in `main` and then pass it to another method, `setIndexZeroToZero`:

Listing (Java) 2.3: Code to change index 0 to the value 0

```java
public static void setIndexZeroToZero(int[] array) {
    array[0] = 0;
}

public static void main(String[] args) {
    int[] arr = {5,5,6,6};
    System.out.println(Arrays.toString(arr));
    // prints [5, 5, 6, 6]

    setIndexZeroToZero(arr);
    System.out.println(Arrays.toString(arr));
    // prints [0, 5, 6, 6]
}
```

Because the memory location of the array `arr` was passed to `array`, the method `setIndexZeroToZero` was dealing with the same object.

Keep in mind that some objects are immutable, like any `String`. This means you can't actually change them. Operations that seem like they change them like replacing part of a string or converting things from upper case to lower case are all returning a newly generated string.

2.3.2 How it works in Python

Practically speaking , everything in Python works the same as in Java. Everything in Python is an object (including the integers, which are immutable.), so when things are passed or assigned into variables, the variable stores the memory location, or *reference*, to the object. Thus when you pass in a variable to a function, the function receives the memory location of the object; data is never duplicated.

Chapter 3

The Array

3.1 Why Arrays

Since this is a data structures course, I assume that students have had exposure to arrays or array like objects. This chapter goes into a bit of a deeper detail that may have been glossed over and introduces the topic in the appropriate language if need be. In other words, I assume you know what an array is and that you've used it to solve problems; writing something like a function to find the average of an array of integers in the language of your choice should be laughably straightforward at this point. However, I don't necessarily assume you know how to use it in Java or Python[1], nor have you "peeked under the hood," so to speak.

There are two other important bits. The first is I want you to understand how your programming language of choice finds an arbitrary index in an array. The second is that I want to hammer in the point that **everything is an object in Java and Python, except they're actually references to objects.**

3.2 Java and Arrays

The Array is a built in class in Java, but the syntax is a bit unique [2]

To create an array in Java we do:

```
Type[] myArray = new Type[sizeOfArray]
```

Here, every item in the array is of whatever `Type` we want, which could be a Class or primitive. Arrays can be whatever integer size we desire, but once set it cannot be changed. This is because to create an array, the computer allocates a contiguous block of memory. If we wanted to resize it, there is no guarantee that this chunk of memory won't have things directly before or after it, preventing us from safely extending its range.

This small fact can lead to some fun shenanigans in other programming languages such as C.

[1] Although we use lists in python

[2] Enough so that I constantly had to look up how to do it my first two years of undergraduate studies, so don't feel too bad if you have to do the same.

3.3 Python and Arrays

Python doesn't really do arrays in the same way. It instead uses Lists, as we'll see in Chapter 5.

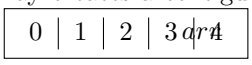
The Python code `myNotArray = []` does not actually make an array like you assume it would in some other language. Instead it makes a list (specifically an arraylist) to contain these items. The syntax works exactly like an array in other languages, but you get access to some nifty operations in Python, like slicing, concatenation, and built-in methods. In addition, Python dynamically resizes this array if we need it bigger or smaller.[3]

However, if you really want or need to use an array in python, you can. There are two ways to accomplish this. The first way is the built in `array` package. This builds a wrapper for the more primitive but efficient C-based array. The python package `numpy` contains yet *another* type of array, this time much more focused on mathematical operations. In short, if you're working in python, use a the default list where you would normally use an array unless you know you should use something more specialized.

Regardless, the next sections will still be valuable because an array-based list, like what Python uses, uses an array internally.

3.4 How an Array Works

As previously mentioned, an array creates a contiguous block of memory. But what does this actually mean?



Here, *arr* does not contain the array; it holds a reference to the array. The correct term varies on the language you are using, but the point is that *arr* tells you the location of the array rather than holding the array itself. Remember, arrays are objects; any variable holding an object is, in reality, holding the memory location of that object.

3.4.1 Operations

To review, arrays have two operations and one attribute: storing a value at an index, retrieving a value from an index, and their size. Interestingly enough, this is one of the few consistent notations across multiple programming languages. For an array `arr`, retrieving a value from an array is and storing it in some variable is done with `myVar = arr[index]`. To store some `newItem` in `arr`, you use `arr[index] = newItem`. Figuring out the length of an array in Java is done with `arr.length`,[4] and in Python, a simple `len(arr)` works.

3.4.2 Array Internals and the Memory Formula

So how does an array actually work? How do you actually retrieve a value from an index? The most crucial thing to keep in mind in this textbook is when you see something like the code below:

[3] We cover the specifics in Chapter 5

[4] This is one of the little things in Java that can be a source of frustration. Strings use `.length()`, arrays use `.length`, and Collections like Lists and Sets use `.size()`. I understand why, but I die a little inside every time I have to explain it.

```
variable = expression;
```

The left side is always a variable. The expression on the right side always yields some memory location.[5] This means that `int[] numbers = new int[10]` stores a memory location in `numbers`. It does not store 10 integers in `numbers`. It only tells you where to find them. Specifically, `numbers` stores the memory location of index 0 of the array. This is true for not only Java, but C as well, and almost every programming language[6].

Thus, the variable that holds the reference to the array is always holding the location of the first index - index 0. In addition, we always know the size of an individual "slot" in an array, either because it is an array of primitives or objects(see below). As a result, our programs can find the memory location of any index of an array using a constant time[7] lookup using the following formula.

$$\texttt{location} = \texttt{arr} + \texttt{index} \cdot \texttt{sizeof(element)}$$

In the above formula, `arr` is the variable holding the reference to the array, which is the starting memory location. Next, `index` is the desired index and `sizeof(element)` is the size (in bytes) of a "slot" of `arr`.

Here's an example. Suppose we have some arr `arr`, which is a reference to an array of 64-bit floating point numbers (`double` in Java, `float` in Python[8]). 64-bits is 8 bytes, so each "slot" of the array is 8 bytes. Let's say that `arr`'s memory location is at address[9] 0x0000BEEF, which means index 0 is at 0x0000BEEF. If we want to find index 3, we can plug it into the formula.

$$\begin{aligned}
\texttt{location} &= \texttt{arr} + \texttt{index} \cdot \texttt{sizeof(element)} \\
&= \texttt{0x0000BEEF} + 3 \cdot \texttt{sizeof(double)} \\
&= \texttt{0x0000BEEF} + 3 \cdot 8 \\
&= \texttt{0x0000BEEF} + 24 \\
&= \texttt{0x0000BEEF} + \texttt{0x00000018} \\
&= \texttt{0x0000BF07}
\end{aligned} \tag{3.1}$$

What if we aren't dealing with primitives, but with objects like Strings instead? In this case, each slot in the array doesn't hold the object itself but instead *a reference to that object*. Thus, each slot needs to be big enough to hold a memory address, ie 32 bits or 64 bits depending on the machine and language.

[5] except for primitives, like `int` in Java

[6] Python and other interpreted languages are slightly more complicated because we are dealing with array lists, thus one additional level of abstraction, so this storage just happens a layer deeper. Esoteric languages like *ook* and *Malbolge* prevent me from making a statement like "all languages."

[7] See Chapter 4 for that term, but basically, no matter how big the array, an index lookup requires only a single multiplication and addition operation.

[8] stings like a C.

[9] When dealing with memory locations/addresses, convention is to use hexadecimal or base 16. If you're unfamiliar, this just means each digit can have 16 distinct symbols, representing the decimal values 0-15, with A being 10 and F being 15.

3.5 Common Array Algorithms

Once again, this chapter isn't designed to teach you how to use arrays or how to solve these simple problems. You have already done that. I present this problems for a few reasons:

- For comparison with Lists in Part II
- You may be learning Java or Python, while knowing the other or neither language.
- We will reexamine these problems in the context of runtime analysis.

3.5.1 Finding Values in an Array

Finding the Minimum

Hopefully you know this one by now! Simply assume the first item is the smallest item, then check it against every other item in the array. If an item is smaller that the current smallest item, it replaces the smallest item.

Listing (Java) 3.1: Finding the Minimum of An Array

```java
public static int findMin(int[] arr) {
    int smallest = arr[0];
    // int smallestIndex = 0;
    for (int i = 1; i < arr.length; i++) {
        if(arr[i] < smallest){
            smallest = arr[i];
            // int smallestIndex = i;
        }
    }
    return smallest; // or smallestIndex
}
```

As seen in the comments, it is fairly straightforward to change this code to return the index, rather than the item.

Listing (Python) 3.2: Finding the Minimum of An Array 2

```python
def findMin(arr):
    smallest = arr[0]
    for num in arr:
        if num < smallest:
            smallest = num
    return smallest
```

Realistically, just use `min(arr)`. If you want to find the index, loop thru using `enumerate`.

Finding the Average

Another common problem. Sum up all the numbers and divide that by how many numbers there were.

Listing (Java) 3.3: Finding the Average in Java

```java
public static double getAverage(int[] arr) {
    int total = 0;
    for(int num : arr) {
        total += num;
    }
    return ((double) total)/arr.length;
}
```

Listing (Python) 3.4: Finding the Average in Python

```python
def getAverage(nums):
    total = 0
    for num in nums:
        total += num
    return total/len(nums)
```

Or just `sum(nums)/len(nums)` for a one liner and impress your classmates with how little code you write.

3.5.2 Limitations

Arrays are awesome solutions for many problems, but they are lacking in ability for some problems. Consider the following exercise:

```
Given a string of text, determine what the most common character of text is.
```

Unless you've seen this problem before, there is no obvious solution. Considerable thought eventually lands on an idea: characters are just integers, so we could assign each one of the characters an index and increment the index each time we see the character.

Listing (Java) 3.5: Most Frequent ASCII Character

```
public static char mostFrequent(String text) {
    int[] tally = new int[128];
    for(char c : text.toCharArray()) {
        tally[(int) c] += 1;
    }

    indexWithHighest = 0;
    for(int index = 0; index<128; index++) {
        if( tally[index] > tally[indexWithHighest]){
            indexWithHighest = index;
        }
    }
    return (char) indexWithHighest;
}
```

If this code is a complete mystery to you, that's okay; this conversion is a bit niche. Every `char` is actually an `int` in disguise. You can read up a little on ASCII and Unicode and then do the following two things. First, try running `System.out.println((int) 'a');`. Casting the `char 'a'` to an `int` gives us the ASCII value 97. We can even do some math with this. Next, try running the code `int what = 'b' - 'a';` and print out the value. You'll get 1, which makes sense, since `b` is one letter after `a`.

Now that we have established every `char` has an `int` value, our code uses this to create an array where the index is the ASCII value of a the character with that `int` value and the `int` stored in that slot is the number of times we've seen that `char` so far.

Listing (Python) 3.6: Most Frequent ASCII Character

```
def mostFrequent(text):
    tally = [0]*128
    for letter in text:
        # https://docs.python.org/3/library/functions.html#ord
        tally[ord(letter)] += 1

    indexWithHighest = 0
    for index, count in enumerate(tally):
        if index> indexWithHighest:
            indexWithHighest = index

    return chr(indexWithHighest) # builtin, reverse of ord
}
```

As above in the Java block, if this code is a complete mystery to you, that's okay; Every glyph is actually an `int` in disguise. You can read up a little on ASCII and Unicode and then play around with `ord` and `chr`. `tally[ord(letter)] += 1` takes the glyph and translates it into it's respective numerical value. As with the Java version, each index in `tally` corresponds to a glyph's numerical value and stores how many times we've see that specific glyph.

However, this has some serious limitations. For one, this breaks if we are not using ascii. What if the text is "こんにちは" or other non-english text? You could create a larger array for all 100000+ unicode characters, but this begins to become less and less feasible. And now what if we change the problem to:

`Given a string of text, determine what the most common word is.`

This suddenly becomes an extremely annoying problem to solve with just arrays[10]. We will solve this problem when we visit Maps in Chapter 14, which are much better suited for this job than arrays.

The other limitation of arrays that their size is immutable. Once an array has been declared, we cannot change its size. This is rather inconvenient for a number of applications where we may not know how many items to store. This will be the focus of our first new data structure: The List.

[10] Those of you coming from Python can stop shouting "use dictionaries!" at the top of your lungs.

Chapter 4

Analyzing Algorithms

Or we would look at the list, but we need to talk about Math. Sorry for the bait-and-switch, but it will make sense shortly.

You don't need much math to be a good programmer, but if you want to be an amazing programmer, you probably need math or very math adjacent skills.

4.1 Cost

Every function, operation, algorithm, or what have you that a computer performs has a *cost*. In fact, there are always multiples costs; we often just focus on the most important one or two costs.

What is most important depends on context. However, in the vast majority of cases, the most important cost to focus on is **time**. When our program is eating away at our storage resources like a hungry child slurping up spaghetti, we can always go out and buy more memory/storage/RAM. If our program requires a large amount of energy consumption, energy is readily available from a variety of sources: batteries, power plugs, internal combustion engines, the giant fusion reactor in the sky to name a few. When we measure cost, we need to do abstractly. Since all computers have different configurations of hardware and software, we look at the number of operations that will be executed, not the overall elapsed time.

4.1.1 Time

A time cost is a measure of not just how long it takes a program to finish executing, bit also how the length of execution is affected by adding additional item.

Time is almost always *the most important cost*. We cannot got out and buy another weeks worth of time. You can't hand a bunch of money to the reaper and ask for a deferral. You can't buy another minute to spend with your mother[1]. Yes, processors get faster as technology marches on, but they get faster slowly and Moore's law ostensibly has its limits. The only way to make our programs realistically run faster is to make them more efficient. And **Big O notation** is the way we will be measuring that efficiency.

[1] Call your mother. She would love to hear from you.

4.1.2 Space

For data structures, we will be measuring their space efficiency in terms of *auxiliary* cost, in other words, how much extra space we need to use over the space used for the items itself. To clarify, any data structure that contains n items of roughly the same size will use $n \times$ `sizeOf(item)` space at minimum, no matter what data structure we use.

4.1.3 Energy

While not something this book concerns itself with, some programmers need to be wary of the amount of energy an algorithm consumes. If energy is expensive and/or battery life needs to be conserved, then choosing an energy efficient algorithm might be a better choice, even if the time or space complexity is higher. Some examples where energy use is a large concern is Mobile Ad-Hoc Networks (MANETs) and battery powered cameras.

4.2 What is Algorithm Analysis

It is very common for beginning computer science students to compare their programs with one another. You may also have noticed that it is common for computer programs to look very similar, especially the simple ones. An interesting question often arises. When two programs solve the same problem but look different, is one program better than the other?

In order to answer this question, we need to remember that there is an important difference between a program and the underlying algorithm that the program is representing. As you have learned, **an algorithm is a generic, step-by-step list of instructions for solving a problem**. It is a method for solving any instance of the problem so that given a particular input, the algorithm produces the desired result. A program, on the other hand, is an algorithm that has been encoded into some programming language. There may be many programs for the same algorithm, depending on the programmer and the programming language being used.

To explore this difference further, consider the functions shown in Listing 4.1 and 4.2. This function solves a familiar problem, computing the sum of the first n integers. The algorithm uses the idea of an accumulator variable that is initialized to 0. The solution then iterates through the n integers, adding each to the accumulator. You may have also heard this referred to as a "running sum."

Listing (Java) 4.1: Sum Of N

```java
public class FindSum {
    public static long sumOfN(int n) {
        long theSum = 0;
        for (int i = 1; i <= n; i++) {
            theSum = theSum + i;
        }
        return theSum;
    }
}
```

Listing (Python) 4.2: Sum of N

```python
def sumOfN(n):
    theSum = 0
    for i in range(1, n + 1):
        theSum = theSum + i
    return theSum
```

Now look at the functions in Listing 4.3 and 4.4. At first glance it may look strange, but upon further inspection you can see that this function is essentially doing the same thing as the previous one. The reason this is not obvious is poor coding. We did not use good identifier names to assist with readability, and we used an extra assignment statement that was not really necessary during the accumulation step.

Listing (Java) 4.3: Sum of N 2 - Obsfucated

```java
public class FindSum2 {
    public static long foo(int tom) {
        long fred = 0;
        for (int nancy = 1; nancy <= tom; nancy++) {
            long joanne = nancy;
            fred = fred + joanne;
        }
        return fred;
    }

    public static void main(String[] args) {
        System.out.println(foo(10));
    }
}
```

Listing (Python) 4.4: Sum of N 2 - Obsfucated

```
def foo(tom):
    fred = 0
    for bill in range(1, tom + 1):
        barney = bill
        fred = fred + barney

    return fred

print(foo(10))
```

The question we raised earlier asked whether one method is better than another. The answer depends on your criteria. The function `sumOfN` is certainly better than the function `foo` if you are concerned with readability. In fact, you have probably seen many examples of this in your introductory programming course since one of the goals there is to help you write programs that are easy to read and easy to understand. In this course, however, we are also interested in characterizing the algorithm itself. (We certainly hope that you will continue to strive to write readable, understandable code.)

Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. We want to be able to consider two algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer. From this perspective, the two methods above seem very similar. They both use essentially the same algorithm to solve the summation problem.

At this point, it is important to think more about what we really mean by computing resources. There are two different ways to look at this. One way is to consider the amount of space or memory an algorithm requires to solve the problem. The amount of space required by a problem solution is typically dictated by the problem instance itself. Every so often, however, there are algorithms that have very specific space requirements, and in those cases we will be very careful to explain the variations.

As an alternative to space requirements, we can analyze and compare algorithms based on the amount of time they require to execute. This measure is sometimes referred to as the execution time or running time of the algorithm. One way we can measure the execution time for the function sumOfN is to do a benchmark analysis. This means that we will track the actual time required for the program to compute its result. In most languages, we can benchmark a function by noting the starting time and ending time of the program or function we are executing. For Java, the `System` module contains a method called `nanoTime` that will return the amount of time that the Java virtual machine has been running, in nanoseconds. For Python, the `time` module has a `time` function[2] which returns the number of seconds since January 1, 00:00:00 UTC.[3]

By using either of these tools to track the start and finish times and then computing the difference, we can get the number of seconds (fractions in most cases) for execution.

[2] I love Python, but sometimes the repetitive naming is vexing.

[3] This is commonly referred to as Unix time and is extremely common.

Listing 4.5 and 4.6 lets you enter the number you want to sum up to n. It then invokes the sumOfN method 25 times and calculates the time required for each trial:

Listing (Java) 4.5: Timing Sum of N in Java

```
import java.util.Scanner;

public class Timing {
    public static long sumOfN(long n) {
        long theSum = 0;
        for (int i = 1; i <= n; i++) {
            theSum = theSum + i;
        }
        return theSum;
    }

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        System.out.print("Find sum from 1 to n: ");
        long n = input.nextInt();

        for (int trial = 0; trial < 25; trial++) {
            long startTime = System.nanoTime();
            long result = sumOfN(n);

            double elapsed = (System.nanoTime() -
            ↪  startTime) / 1.0E9;
            System.out.printf("Trial %d: Sum %d: time %.6f
            ↪  sec.%n",
            trial, result, elapsed);
        }
    }
}
```

Listing (Python) 4.6: Timing Sum of N in Python

```
import time

def sumOfN(n):
    start = time.time()

    the_sum = 0
    for i in range(1, n + 1):
        the_sum = the_sum + i

    end = time.time()

return the_sum, end - start
```

Listing 4.5 shows the original sumOfN function with the timing calls embedded before and after the summation. The function returns a tuple consisting of the result and the amount of time (in seconds) required for the calculation. Here is start and end of the output when we enter 10000 for the limit of the sum using the Java code:

```
Trial 0: Sum 50005000: time 0.003886 sec.
Trial 1: Sum 50005000: time 0.004009 sec.
Trial 2: Sum 50005000: time 0.000186 sec.
Trial 3: Sum 50005000: time 0.000185 sec.
...
Trial 20: Sum 50005000: time 0.000125 sec.
Trial 21: Sum 50005000: time 0.000124 sec.
Trial 22: Sum 50005000: time 0.000125 sec.
Trial 23: Sum 50005000: time 0.000124 sec.
Trial 24: Sum 50005000: time 0.000124 sec.
```

Why does the time go down from .003886 seconds to .000124? Because the Java Virtual machine and the computer hardware itself cache results, keeping them in memory for future access. We run the trial loop 25 times in order to give the cache time to stabilize.[4]

We discover that the time is fairly consistent and it takes on average about 0.000125 seconds to execute that code. What if we run the function adding the first 100,000 integers? (Showing only the final five trials)

```
Trial 20: Sum 5000050000: time 0.001225 sec.
Trial 21: Sum 5000050000: time 0.001226 sec.
Trial 22: Sum 5000050000: time 0.001225 sec.
Trial 23: Sum 5000050000: time 0.001224 sec.
Trial 24: Sum 5000050000: time 0.001224 sec.
```

Again, the time required for each run, although longer, is very consistent, averaging about 10 times more seconds. For $n = 1,000,000$ we get:

[4]The Python results are similar, minus the caching.

```
Trial 20: Sum 500000500000: time 0.012350 sec.
Trial 21: Sum 500000500000: time 0.012411 sec.
Trial 22: Sum 500000500000: time 0.012353 sec.
Trial 23: Sum 500000500000: time 0.012443 sec.
Trial 24: Sum 500000500000: time 0.012447 sec.
```

In this case, the average again turns out to be about 10 times the previous experiment.

Now consider Listings 4.7 and 4.8, which shows a different means of solving the summation problem. This method, sumOfNImproved, takes advantage of a closed equation $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{(n)(n+1)}{2}$ to compute the sum of the first n integers without iterating[5].

Listing (Java) 4.7: An Improvement To Constant Time

```java
public static long sumOfNImproved(long n) {
    long theSum = n * (n + 1) / 2;
    return theSum;
}
```

Listing (Python) 4.8: An Improvement To Constant Time

```python
def sumOfNImproved(n):
    return (n * (n + 1)) / 2
print(sumOfNImproved(10))
```

If we do the same benchmark measurement with this revised code, using five different values for n (10,000, 100,000, 1,000,000, 10,000,000, and 100,000,000), we get the following results from averaging the last five trials:

```
Sum 50005000:        time 0.0000088 sec.
Sum 5000050000:      time 0.0000092 sec.
Sum 500000500000:    time 0.0000082 sec.
Sum 50000005000000: time 0.0000078 sec.
```

There are two important things to notice about this output. First, the times recorded above are shorter than any of the previous examples. Second, they are very consistent no matter what the value of n. It appears that `sumOfNImproved` is hardly impacted by the number of integers being added.

But what does this benchmark really tell us? Intuitively, we can see that the iterative solutions seem to be doing more work since some program steps are being repeated. This is likely the reason it is taking longer. Also, the time required for the iterative solution seems to increase as we increase the value of n. However, if we ran the same function on a different computer or used a different method language, we would likely get different results. It could take even longer to perform `sumOfNImproved` if the computer were older.

We need a better way to characterize these algorithms with respect to execution time. The benchmark technique computes the actual time to execute. It

[5]This sequence of numbers is the **Triangular Number Series** and shows up a lot in analysis.

does not really provide us with a useful measurement because it is dependent on a particular machine, program, time of day, compiler, and programming language. Instead, we would like to have a characterization that is independent of the program or computer being used. This measure would then be useful for judging the algorithm alone and could be used to compare algorithms across implementations.

4.3 Big O Notation

When trying to characterize an algorithm's efficiency in terms of execution time, independent of any particular program or computer, it is important to quantify the number of operations or steps that the algorithm will require. If each of these steps is considered to be a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. Deciding on an appropriate basic unit of computation can be a complicated problem and will depend on how the algorithm is implemented.

A good basic unit of computation for comparing the summation algorithms shown earlier might be the number of assignment statements performed to compute the sum. In the function sumOfN, the number of assignment statements is 1 ( ) plus the value of n (the number of times we perform). We can denote this by a function, call it , where . The parameter n is often referred to as the "size of the problem," and we can read this as "is the time it takes to solve a problem of size , namely steps."

In the summation functions given above, it makes sense to use the number of terms in the summation to denote the size of the problem. We can then say that the sum of the first 100,000 integers is a bigger instance of the summation problem than the sum of the first 1,000. Because of this, it might seem reasonable that the time required to solve the larger case would be greater than for the smaller case. Our goal then is to show how the algorithm' s execution time changes with respect to the size of the problem.

Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the function. In other words, as the problem gets larger, some portion of the function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The order of magnitude function describes the part of that increases the fastest as the value of n increases. Order of magnitude is often called Big O notation (O stands for order) and written as . It provides a useful approximation of the actual number of steps in the computation. The function provides a simple representation of the dominant part of the original .

In the above example, . As n gets larger, the constant 1 will become less and less significant to the final result. If we are looking for an approximation for , then we can drop the 1 and simply say that the running time is . It is important to note that the 1 is certainly significant for . However, as n gets large, our approximation will be just as accurate without it.

As another example, suppose that for some algorithm, the exact number of steps is . When n is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function. However, as n gets larger, the term becomes the most important. In fact, when n is really large, the other two terms become

insignificant in the role that they play in determining the final result. Again, to approximate as n gets large, we can ignore the other terms and focus on . In addition, the coefficient becomes insignificant as n gets large. We would say then that the function has an order of magnitude , or simply that it is .

Although we do not see this in the summation example, sometimes the performance of an algorithm depends on the exact values of the data rather than simply the size of the problem. For these kinds of algorithms we need to characterize their performance in terms of best-case, worst-case, or average-case performance. The worst-case performance refers to a particular data set where the algorithm performs especially poorly, whereas a different data set for the exact same algorithm might have extraordinarily good (best-case) performance. However, in most cases the algorithm performs somewhere in between these two extremes (average-case performance). It is important for a computer scientist to understand these distinctions so they are not misled by one particular case.

A number of very common order of magnitude functions will come up over and over as you study algorithms. These are shown in Table 2.3.1. In order to decide which of these functions is the dominant part of any function, we must see how they compare with one another as n gets large.

TABLE GOES Here

Figure 2.3.2 shows graphs of the common functions from Table 2.3.1. Notice that when n is small, the functions are not very well defined with respect to one another. It is hard to tell which is dominant. However, as n grows, there is a definite relationship and it is easy to see how they compare with one another.

PLOT OF COMMON FUNCTIONS GOES HERE

As a final example, suppose that we have the fragment of Java code shown in Listing 2.3.3. Although this program does not really do anything, it is instructive to see how we can take actual code and analyze performance.

```
int a = 5;
int b = 6;
int c = 10;
int n = 1000;
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                int x = i * i;
                int y = j * j;
                int z = i * j;
        }
}

for (int k = 0; k < n; k++) {
        int w = a * k + 45;
        int v = b * b;
}
int d = 33;
```

Listing 2.3.3.

The number of assignment operations is the sum of four terms. The first term is the constant 4, representing the four assignment statements at the start of the fragment. The second term is , since there are three statements that are performed times due to the nested iteration. The third term is , two statements

iterated n times. Finally, the fourth term is the constant 1, representing the final assignment statement. This gives us . By looking at the exponents, we can see that the term will be dominant and therefore this fragment of code is . Note that all of the other terms as well as the coefficient on the dominant term can be ignored as n grows larger.

GRAPH COMPARING T(N)

Figure 2.3.4 shows a few of the common Big O functions as they compare with the function discussed above. Note that is initially larger than the cubic function. However, as n grows, the cubic function quickly overtakes . We can also see that then follows the quadratic function as continues to grow.

- What is big O
- how to read it
- Aside about big omega and theta
- How wrong usage annoys mathematician
- refers to cost in general, but used for time usually
- space complexity
- Common runtimes
- runtimes we"ll focus on now
- runtimes we focus on later

Annoying Your Friends in Math

Programmers are often loose with their language with Big O notation and often refer to runtime using Big O notation, imposing an upper bound on runtime. They do this even when they ar

Another common mistake about Big O is to assume Big O means worst case scenario and Big Omega is best case. This is not so. Big O and the rest are *tools* to describe best, worst, or average case.

4.3.1 Common Runtimes in this book

While we've introduced many different runtimes, not all occur at the same level of frequency.

4.3.2 Space Complexity

4.4 Examples with Arrays

- Retrieval - refer back to earlier chapter for address lookup - SHow how that is constant time.
- Replacement
- Linear Search
- Binary Search

Part II

Lists

Chapter 5

Array Lists

The first data structure we will be studying is the list. The list is by far the most relatable data structure, as humans deal with lists on a regular basis.

5.1 What is a List?

When you get right down to it, lists are defined by order. We don't have to take advantage of this order, but its there. Populated lists have a first item and they have a last item.

Let's start by looking at two non-computer examples of lists. Take a look at this quest below from a hypothetical fantasy game:

Quest: Slay the Dragon of Doom

- Get Sword of Dragonslaying
- Locate the map to Dragon Lair of Doom
- Travel to the Dragon Lair of Doom
- Slay the Dragon of Doom
- Return to the Castle

Here, the order is implied by the contents of the list - you can't beat the dragon without the macguffin and you certainly can't fight it without being able to find it. Generally speaking, going up against a dragon without any preparation is foolhardy in the extreme, but I digress.

Thus, you must get the special sword[1] first, and you must get the map to find the lair before you can physically travel there.

[1]What if its possible to get the map before the sword? We'll see much later this kind of quest and it's requirements are much better handled by a directed acyclic graph in Chapter 17, but this example is fine for teaching lists.

shopping list example

While lists are defined by order, we don't necessarily ascribe any meaning to the order. Take a look at the shopping list below:

<Shopping List>

While bread is the first item on this list, being the first item in the shopping list in this case has no special meaning. It's not the most important item on the list[2], nor is it necessarily the item I'm going to pick up first. Thus, as previously stated, while lists have an order, the whether or not order is important depends on the context.

5.2 Why Should I care

Where arrays and lists differ is that lists can grow to an arbitrary size, whereas arrays are static. Arrays can't get bigger, lists can.In this chapter, we will be discussing the **array list**, the first of two types of lists this book will teach you.

A note on terminology

An **array list** is a type of list. These are sometimes called dynamic arrays.

As mentioned in Section **??**, Python doesn't have arrays. If you've been programming in Python, you've been using an array list the entire time you've declared `[]`. They are usually just called lists rather than array lists for simplicity's sake. I will be using the Java nomenclature for the majority of the book as this allows me to be clear about the types and implementations of data structures.

5.2.1 Lists in Java

An array list in Java is represented by the class `ArrayList`. Here is an example:

An Aside about interfaces

This textbook assumes that you have already taken your requisite object oriented programming course, but in case you haven't or it's been a while, I'll review briefly here.

An interface is about as abstract as a class can get and ties deeply to how Java deals with polymorphism. In fact, an interface contains only abstract methods, which must be implemented by the inheriting class.

What about python? Python deals with polymorphism using duck typing, originating from the idiom "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"

[2] obviously, that's the cookies

Here is the source code

5.3 Generics

You may have noticed that when we create the arraylist

5.3.1 What are they?

Before we get to deep into lists, we need to have a discussion about generics. Generics are a way of restricting and specifying what types can go into a collection.

5.3.2 But Why?

Using generics has two big purposes: strong typing and the lack of need for casting. This textbook deals with handling collections of data. In fact, the Java superclass for a lot of the topics we cover is called a `Collection`. Generics allow us to predefine what precisely the collection will hold. If we do not do so, the only thing we can assume a collection holds is objects of type `Object`.

This creates two big headaches

- In strongly typed languages like Java, we will need to cast objects to the appropriate type when extracting them. In duck-typed languages like Python, we rely on thoughts and prayers.
- There are no safeguards to prevent us from inserting items of the wrong type into the collection.

5.4 List Operations

5.4.1 Size

We need some easy way of knowing how big our lists are, if for no other reason than to make sure our add and remove methods can figure our their valid indices.

5.4.2 Add

By default, we add items to the end of the list, but we can also add items to any index we want.

When we add an item at some specific index i, the item at i and all indices to the right shift over one. In other words, what was at i is now at $i + 1$, what was at $i + 1$ is at $i + 2$, and so on.

This also an understandable restriction to adding items to a list - we cannot an item to any index greater than `myList.size() +1`. Anything greater wouldn't be at the end of the list; it would be beyond it. The same goes for negative indices[3].

<possible picture showing a legal an illegal add>

We will cover this operation in more detail when we implement the add method for the arraylist

[3]Python does allow negative indexes, but we will ignore that for now

5.4.3 Remove

We can remove items from a list much in the same way we can add them. When removing an item at index i, the

For example, in the image below, we are removing the item at index 3, the word "cookie," from the list.

<image before>

C is for cookie that's good enough for me

<image after>

C is for that's good enough for me

5.4.4 Get

Get is how we retrieve our items from the list. Given an index, get will give us the value that has been stored at that index.

5.4.5 Set

5.5 ArrayLists

An array list, as you might have guessed, are lists built using *arrays*.[4] They work by growing or shrinking the array[5] automatically as items are added or removed from the list, giving the illusion that the data structure can hold an arbitrary amount of data.

We'll go into the specifics of how this works in Section 5.7.

Java's ArrayLists

Java's arrayList

Python's Lists

Python's lists, such as below:

```
l = [1,2,3] # this is a list, not an array!
```

are actually array lists!

Python uses a different vocabulary for some of the methods we'll be implementing below. For example, take the action of adding an item to a list. Python uses the `append` method to add an item to end of the list and `insert` to put an item into the middle of the list. Java (who's vocabulary we'll be following), uses `add` for both these contexts.

5.6 Example Algorithms

```
public static <E> boolean isPermutation(List<E> listA, List<E> listB) {

        if(listA.size() != listB.size()) {
```

[4] Shockingly, many of the names we give things at this point actually make sense.

[5] A lie. As you'll see we don't actually change the size of an array; we create a new array of the appropriate size and copy everything over

```
                return false;
        }
        for(int i  = 0; i < listA.size() ; i++){
                E item =  listA.get(i);
                int countA = 0;
                int countB = 0;

                for (E element : listA) {
                        if(item.equals(element)){
                                countA++;
                        }
                }
                for (E element : listB) {
                        if(item.equals(element)){
                                countB++;
                        }
                }
                if(countA != countB) {
                        return false;
                }
        }
        return true;
}
```

5.7 Building an ArrayList

To truly understand how a data structure works we need to implement it ourselves. We will be making simpler versions of what's actually implemented in the language of your choice, but the logic and obstacles we need to overcome are the same.

5.7.1 Caveats

MyArrayList.java

We will not be implementing the `List` interface. We don't need to implement all the functions to get an understanding of how the fundamentals of an arraylist work. Implementing the list interface would take up a hideous amount of paper and get in the way of actually understanding the code. If you want to see how `ArrayList` actually looks, you can look at the javadoc.

myArrayList.py

For python, this will require some suspension of disbelief, as our array list will require using an array, and as previously discussed, arrays are shirked in favor of arraylists in python. We'll be using a list and pretending it's an array. Silly? Yes. But it will keep our code compact and easier to understand.

5.7.2 Instance Variables

Believe it or not, we only need to keep track of three instance variables to get our arraylist working.

theData We need an array to actually store the items. This is it.

size Size here refers to the total number of items we have stored in the array.

capacity This is the number of items the underlying array in our list can hold. It is the maximum size of the list before we have increase the capacity and move everything `theData` to a new array of length `capacity`. This is not strictly necessary as we can get it by querying `theData`'s length. However, making it it's own variable will help with the readability.

It is very easy to confuse size and capacity since they both deal with counting how many elements. When I talk about `size`, I am talking about the number of items we have stored in the list we are making. Capacity, on the other hand, depends on the length of the built-in array.

Java

Listing (Java) 5.1: The Beginning

```java
public class MyArrayList<E> {

    private E[] theData
    private int size;       // how many items in the list
    private int capacity;   // how many items the underlying
    ↪  array can hold
```

First, note the `<E>` after `MyArrayList`. This means that we're saying:

- MyArrayList is designed to hold a specific type of object.
- Every `E` we see is a placeholder for some type, which will be that same across the entire lifespan of the object.

Python

In python, we will be creating our instance variables in the constructor below. We will end up with this at the end of Section 5.7.3.

Listing (Python) 5.2: The Beginning

```python
class MyArrayList(object):
    def __init__(self):
        self.size = 0
        self.capacity = 10
        self.theData = [None]*self.capacity
```

5.7.3 Constructor

We need to set the variables to their initial values upon creating the arraylist. The `size` will be 0, since we won't have any objects stored in it yet. We will set the `initial` capacity to 10, as this is the default behavior of Java's ArrayList class. It's a small number and thus won't create much wasted space if we don't fill up `theData`. `theData` will be an empty array of `capacity` length. If `theData` becomes full, we will create a bigger array to hold our items using the `reallocate()` method (Section **??**)

Java

With our constructor, we have one line of weird black magic in order to create an Array of `E[]`'s.

```
public MyArrayList(){
        size = 0;
        capacity = 10;
        theData = (E[]) new Object[10]; // this generates a warning
}
```

So what's going on with the last line? Typically, when creating an array, we would just say:

```
//doing this in the constructor gives us an error.
TYPE[] myArray  = new Type[desired_size];
```

However, Java won't let you create new `E` objects since there's no telling what the constructors will be. This rule extends to arrays of `E`, like so:

```
theData =  new E[10];
```

However, when creating a new empty array of objects of any type, we're just making an array of nulls which will eventually be replaced by references to objects. Thus, even though the Java compiler will yell at us about Type safety, we can instead create an array of `Object` and then tell , since all references to any types are the same size.

```
// creating one array of nulls and telling Java
// its another type of array of nulls.
theData = (E[]) new Object[10];
```

Remember how Java and most modern programming languages deal with objects; if you're assigning an object to a variable, like in `Object o = new Object()`, we are storing a reference to that object. Thus, when we add an item to a list, what really happens is we'll be adding a reference to it - the instructions on how to find it in memory.

Python

Python is fairly straightforward, with the caveat that we are pretending `theData` is an array, and not a list.

```
class MyArrayList(object):
        def __init__(self):
                self.size = 0
                self.capacity = 10
                self.theData = [None]*self.capacity
```

Since built-in lists in Python grow and shrink like we would expect a list to, we initialize `theData` with 10 `None` objects[6] to mimic the way an array would be initialized.

5.7.4 Size

Now, we will add a size method to our list; fairly straightforward in `Java`.

```
public int size() {
        return size;
}
```

In Python, we can go ahead and use the built in `__len__` method, which can then be invoked with `len(myList)`.

```
def __len__(self):
        return self.size
```

Retrieving the size of our list is always $O(1)$, as we are just accessing a variable and returning its value.

5.7.5 The Add Method

Now it's time to dig into the bulk of our code: adding items to our list. To do this, I'll be creating two methods: one for adding to the end of the list (an extremely common operation) and one for adding at any index in the list.

In Java, we will overload these two methods and call them both add. We will have an `add( E item )` for adding to the end and an `add(int index, E item)` for every other case. In Python, these two `add` methods are called `append` and `insert` respectively, as Python does not support method overloading.

We will be looking at the case of adding to a specific index first.

Cases

The `add` method has 5 basic parts, only three of which involve actual thinking about how to code:

1. Check index to see if our index in bounds
    - If it is, crash the program.
2. Check to see if our array list has room to add a new item.
    - If there is no room, make some!
    - How we do this is covered in Section **??**.

[6] This is the Python equivalent to the Java `null`.

3. Shift all the existing items from `index` to the end of the list over one index to the right. This moves all the items already in the list to their new locations.

4. Store the item.

5. Increment the size.

Those last two steps are important but not complicated. We will go ahead and handle them now and put in comments for the other parts.

```
public void add(int index, E item) {
        // Check the index

        // do we have room?

        //shift over existing items

        theData[index] = item;
        size++;
}
```

```
def insert(self, index: int, item):
        # Check the index

        # do we have room?

        # shift over existing items

        self.theData[index] =  item
        self.size += 1
```

Checking The Index

An optional step for pedagogy, but good practice. If the index is less than, we reject it. If the `index > size`, we reject it. The case of `index == size` is perfectly fine, but it feels weird, since you should have the rule "valid indexes are 0...`array_size`" carved into your soul by this point. This is because the index `size` would be the next empty slot for use to put an item. Once we insert the item, we increment the size at the step of the method. After that, our rule about valid indexes becomes true again.

```
public void add(int index, E item) {
        // Check the index
        if(index < 0 || index > size) {
                throw new IndexOutOfBoundsException("Index " +index+ " is out of bounds.");
        }

        // do we have room?

        //shift over existing items
```

```
        theData[index] = item;
        size++;
}
```

In python, we take the additional step of checking if the index is an `int`.

```
def insert(self, index: int, item):
# Check the index
        if not isinstance(index, int):
                raise IndexError(index + " is not an integer.")
        if index < 0 or index > self.size:
                raise IndexError("Index " + str(index) +  " is out of range.")
        # do we have room?

        # shift over existing items

        self.theData[index] =  item
        self.size += 1
```

Deciding to Reallocate

Our array is only so big; if our current `size` and `capacity` are the same, we don't have any more room. In this situation, we call `reallocate`, which doubles[7] our capacity. We will solve this issue in Section **??** and handwave the implementation for now.

```
public void add(int index, E item) {
        // Check the index
        if(index < 0 || index > size) {
                throw new IndexOutOfBoundsException("Index " +index+ " is out of boun
        }

        // do we have room?
        if(size == capacity) {
                this.reallocate();
        }
        //shift over existing items

        theData[index] = item;
        size++;
}
```

In python, we take the additional step of checking if the index is an `int`.

```
def insert(self, index: int, item):
        # Check the index
        if not isinstance(index, int):
                raise IndexError(index + " is not an integer.")
```

[7] As we will see later, doubling is what we chose for our implementation, but other options exist.

```
        if index < 0 or index > self.size:
                raise IndexError("Index " + str(index) +  " is out of range.")
        # do we have room?
        if self.size ==  self.capacity:
                self.__reallocate()

        # shift over existing items

        self.theData[index] =  item
        self.size += 1
```

Shifting the Items

As mentioned previously, if `index == size`, we will be inserting the item we want to add into the next unused slot.

Reallocation Implementation

When we need to grow our arraylist, can't actually physically change the size of the array `theData`; you can't change the size of an array. So we cheat. We create a new array twice [8] the capacity of `theData`. We then copy everything over to the new array and then store the reference to that new array in `theData`, making it our new underlying array.

```
private void reallocate(){
        //doubles or 1.5x capacity
        //don't do +1 capacity
        capacity = 2 * capacity;
        E[] newData = (E[]) new Object[capacity];
        for(int i = 0; i < theData.length; i++) {
                newData[i] =  theData[i];
        }
        theData = newData;
}
```

We want to double our capacity or at least increase it by 50%, rather than increasing it by a static number. Consider if we increase the capacity by one each time we reallocated. If we did that, we would have to reallocate every time we added a new item to the list. This would mean that every time we add an item to list, add becomes a linear time - $O(n)$ - operation.

Having empty slots might seem wasteful, but the advantage is that it takes constant time to add to the end of the arraylist. This is because we don't have to shift any existing elements around. It is a classic time/space trade-off.

Because reallocation is a *relatively* rare event compared to adding, we don't typically take that cost into account when analyzing an algorithm with a large number of `add` commands. This is because if we do have some capacity n, in order to trigger reallocation with a runtime of $O(n)$, we have to do n add

[8] The one thing worth noting is that the real implementation of a list in python, listobject.c, uses a completely different pattern than doubling the capacity. This is more complicated than we need for this book; doubling is much simpler and accomplishes what we need.

operations first. We can then "spread out" the cost of the `reallocate` operation over our `add` operations.

Finished Code

```
public void add(int index, E item) {
        if(index < 0 || index > size) {
                throw new IndexOutOfBoundsException("Index " + index + " out of bound
        }

        if(size == capacity) {
                this.reallocate(); // O(n) time...sometimes.  Amortized over the cost
        }

        for(int i = size - 1; i >= index; i--) { //If adding to the end... constant
                E temp = theData[i];  // Store the item from
                theData[i+1] = temp;  // Move the item from
        }

        theData[index] = item;
        size++;
}

private void reallocate(){
        //doubles or 1.5x capacity
        //don't do +1 capacity
        capacity = 2 * capacity;

        E[] newData = (E[]) new Object[capacity];
        for(int i = 0; i < theData.length; i++) {
                newData[i] =  theData[i];
        }

        theData = newData;


}
```

```
def insert(self, index: int, item):
        if not isinstance(index, int):
                raise IndexError(index + " is not an integer.")
        if index < 0 or index > self.size:
                raise IndexError("Index " + str(index) +  " is out of range.")
        if self.size ==  self.capacity:
                self.__reallocate()
        for i in range(self.size -1, index -1, -1):
                temp =  self.theData[i]
                self.theData[i+1] = temp
        self.theData[index] =  item
        self.size += 1
```

```
def __reallocate(self):
        self.capacity = self.capacity * 2
        newData =  [None] * self.capacity
        for index, item in enumerate(self.theData):
        newData[index] = item
        self.theData = newData
```

Adding to the End

As previously mentioned, adding to the end is an extremely common operation, so we will overload our `add` method. If our list is provided with only an `item`, as opposed to an `item` and an `index`, we will just add that `item` to the end. Since we already wrote a perfectly good `add` method already that we know works, we'll just have our new method call that one.

```
public boolean add(E item) {
        this.add(size, item); // size is the last valid index
        return true; // What?
}
```

Why are we returning `true` here? The short answer is practice and consistency with future data structures. The long answer is any `Collection` in Java has must have an `add` method and a `List` is type of `Collection`[9].

`Collection` specifies that `add` must take in an `item` and return a `boolean`. A `true` signals the `add` is successful. A `false` signals that we could not add the `item`. For example, this might happen with a `Set` (Chapter 13)

On the other hand, our Adding at a specific index is unique to lists, and not part of collections, and will always work. Therefore, there's no need to return a boolean.

5.7.6 toString and __str__

Now that we supposedly have a method for adding items into the list, the next step is to test it. The easiest way to test it is by printing out the contents of the list. We'll do this in the laziest way possible.

In java, that would be invoking the `Arrays.toString` method, since directly turning an array into a string gives you representation of the memory location:

```
public String toString(){
        // return theData+""; // memory location

        return Arrays.toString(theData); // import the library
}
```

That said, implementing it ourselves gives us good practice handling a common fence-posting problem, i.e. we need to print n items separated by $n-1$ commas.

[9]Our `MyArrayList` isn't technically a `Collection` since we did not implement the `List` interface, but I digress.

```
public String toString(){
        String output = "["+theData[0];
        for (int i = 1; i < size; i++) {
                output+= ", " + theData[i];
        }


        return output + "]";
}
```

```
def __str__(self): # second attempt
        output =  "["
        #only include indexes from 0 to size-1
        for item in self.theData[:self.size]:
                output += str(item) +","
        output = output[:-1] # remove the last comma
        return output + "]"
```

5.7.7 Get and Set

The `get` and `set` methods are fairly straightforward:

`get` - Given an `index`, retrieve the item stored at that `index`.

`set` - Given an `index`, replace the old item stored at that `index` with the provided `item`.

`set` has one additional quirk, we also want to return the old item we're replacing, just in case the programmer wants to doing something with the old item. This would obviate the need for pairing a `get` and `set` call with each other if we want to replace the old item, but do something else with it.

For both `get` and `set`, we want to throw some kind of error if the provided index is out of bounds.

Java

Our `get` is fairly straightforward, but feel free to give more information with the error.

```
public E get(int index) {
        if(index  < 0 || index >= size) {
                throw new IndexOutOfBoundsException("Index " + index + " out of bound
        }
        return theData[index];
}
```

The same goes for our `set` method.

```
public E set(int index, E item) {
        if(index < 0 || index >= size) {
                throw new IndexOutOfBoundsException("Index " + index + " out of bound
        }
```

```
        E oldItem  = theData[index];
        theData[index] = item;

        return oldItem;
}
```

Python

Python supports negative indices.

We can take advantage of some of the method calls built into python to make our `myarraylist` support indexing.

```
        def __getitem__(self, index):
        if index < 0:
        index =  index % self.size  # yes!
        # If you're confused, test modulo on
        # negative numbers in python.
        if index >= self.size:
        raise IndexError("Index " + str(index) +  " is out of range.")
        return self.theData[index]
```

This method, as written, will return `None` if the user tries to access an index that is within in the bounds of the capacity but above the size. The same thing will happen if we use negative indices.

While this is fine for our pedagogical programming purposes, prudence posits proactive protection. That is to say, we should ask "how do we prevent out users from accidentally getting the wrong data when they should be getting an error."

Below, we will add two index checks.

```
def __getitem__(self, index):
        if index < 0:
                index =  index % self.size  # yes!
                # If you're confused, test modulo on
                # negative numbers in python.
        if index >= self.size:
                raise IndexError("Index " + str(index) +  " is out of range.")
        return self.theData[index]
```

5.7.8 Remove

The code for `remove` is almost identical in structure to `add`, but without a case for checking if there's room. Since we are removing, we don't have to worry about running out of room. We also make sure we save the item we are removing and return it, for the same reason we do for the `set` method.

- Check if `index` is valid.
- Save the item at `index` for later.
- Shift each item to the right of index over (indices greater than `index`) one to the left. This will overwrite what's stored at index, which is why we saved it.

- Decrement the size.
- Return the saved item.

A word of warning with `remove` operations on "real" implementations. Removing items from a list while you are iterating over it has the potential to get messy. Languages can sometimes even throw runtime exceptions to *prevent* you from doing it. See the problem in Section 5.10.1

5.8 Analysis

When reading through our analysis, please keep in mind that we made a number of pedagogical choices when writing our Array List.

We did this to make our code readable and to help gain an understanding of the mechanisms .

The Array List implementation in your language of choice probably has a huge number of optimizations, at the cost of readability and complexity. For example, at the time of writing, `listobject.c`, the source code for the Python list, is almost 3500 lines long [2].

Those caveats aside, lets talk about the four primary operations for Lists that have a cost: add, remove, get, and set.

5.8.1 Add/Remove

The runtime for adda dnd

5.8.2 Get/Set

Arraylists use the same memory formula discussed in Section 3.4.2 to find a specific index. This calculation, which is an addition and multiplcation operation, takes the same amount of time no matter how big the ArrayList is. Thus the runtime is $\Theta(1)$

5.8.3 A Note on Storage

5.9 A Few More Useful Methods

5.9.1 Constructors

Java's `ArrayList` can optionally take in an integer as an argument. This will start the underlying array's length at that value, rather than the default of 10. This is useful if you know exactly how big your List will be. However, if you aren't removing any items when populating your list, consider using an array instead.

5.9.2 Manually Adjusting the Capacity

Java provides two methods for manually adjusting the ArrayList capacity. The method `ensureCapacity(n)` forces the ArrayList to grow to a capacity of `n` items, if it can't already. Conversely The `trimToSize()` shrinks the capacity to

be equal to the current size. This is useful if we know the ArrayList won't get any larger than it currently is and want to eliminate the wasting memory with empty array slots.

Python will automatically optimize lists for you. Python will automatically resize the list to shrink it if necessary [2].

5.9.3 Adding Multiple Items in One Invocation

One common operation is to move or copy all the items from one list to another. In Java, we can use the `addAll()` method, which takes any Java collection as a parameter and all the items in that collection to the object.

```
List<Integer> a =  new ArrayList<>();
List<Integer> b =  new ArrayList<>();
for(int i = 0; i <3; i++) {
        a.add(i);
}
for(int i = 3; i <6; i++) {
        b.add(i);
}
a.addAll(b);
System.out.println(a); // 0 to 5 inclusive
System.out.println(b); // [3, 4, 5]
```

In Python, we can use the `extend()` method on anything that is iterable or use some clever slicing. However, I would always recommend using the method call over the slice, since a method invocation is always more readable.

```
a = [0, 1, 2]
b = [3, 4, 5]
c = a + b      # creates a new list, which is similar to our goal
a.extend(b)    # adds all of b's items to a
a[len(a):] = b # does the same thing but looks gross.
```

A common beginner mistake in Python is to try to extend a list by calling `append` on the list like so.

```
a = [0, 1, 2]
b = [3, 4, 5]
a.append(b) # a is now [0, 1, 2, [3, 4, 5]]
```

This adds the entire list a single item in the list.

5.10 Exercises

5.10.1 Remove All Instances

Write a method called `removeAllInstances()` which takes in a `List` and item[10]. The method then proceeds to remove each item in the list that matches the given item. For example, if the method is passed the `List<Integer> [1, 4, 5, 6, 5, 5, 2]` and the `Integer 5`, the method removes all `5`'s from the `List`. The `List` then becomes `[1, 4, 6, 2]`. It should return nothing, since the changes the `List` it was provided. [11]

[10] In other words, the first parameter is a list of generics and the other input is a single item of the same type the list holds.

[11] This one is extremely tricky, since removing an item shifts the indexes.

5.11 Source Code

5.11.1 Java

```
package arraylists;


// Change this up to be distinct from KW;  been teaching so many years using their text that th
public class MyArrayList<E> {

    private int size;  // how many items are in the list
    private int capacity;  // how many items the underlying array can hold
    private E[] theData;

    public MyArrayList(){
        size = 0;
        capacity = 10;
        theData = (E[]) new Object[10];
    }


    public int size() {  // O(1)
        return size;
    }


    public boolean isEmpty() {
        return (size == 0);
    }

    public boolean add(E item) {
        this.add(size, item);
        return true;
    }


    public void add(int index, E item) {
        if(index < 0 || index > size) {
            throw new IndexOutOfBoundsException("Index " +index+ " is out of bounds.");
        }

        if(size == capacity) {  // O(n) time...sometimes.  Amortized over the cost of adding
            this.reallocate();
        }

        for(int i = size - 1; i >= index; i--) { //If adding to the end... constant
            E temp = theData[i];
            theData[i+1] = temp;
        }

        theData[index] = item;
```

```
        size++;
    }

    private void reallocate(){
        //doubles or 1.5x capacity
        //don't do +1 capacity
        capacity = 2 * capacity;

        E[] newData = (E[]) new Object[capacity];
        for(int i = 0; i < theData.length; i++) {
            newData[i] =  theData[i];
        }

        theData = newData;


    }



    public E remove(int index) {
        if(index  < 0 || index >= size) {
            throw new IndexOutOfBoundsException("WE ALREADY WENT OVER THIS! IT'S OUT
        }
        E item =  theData[index];

        for(int i = index  + 1; i < size; i++) {  //O(n), unless we remove last item
            theData[i-1] =  theData[i];
        }

        size--;
        return item;
    }

    public E get(int index) {
        if(index  < 0 || index >= size) {
            throw new IndexOutOfBoundsException("WE ALREADY WENT OVER THIS! IT'S OUT
        }
        return theData[index];
    }


    public E set(int index, E item) {
        if(index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("WE ALREADY DID THIS JOKE!");
        }
        E oldItem  = theData[index];
        theData[index] = item;

        return oldItem;
    }
```

```
    public int indexOf(E item) {
        for (int i = 0; i < size; i++) {
            if(item.equals(theData[i])){
                return i;
            }
        }
        return -1;
    }


    public boolean contains(E item) {
        for (int i = 0; i < size; i++) {
            if(item.equals(theData[i])){
                return true;
            }
        }
        return false;
    }


    public String toString(){
        String output = "["+theData[0];
        for (int i = 1; i < size; i++) {
            output+= ", " + theData[i];
        }


       return output + "]";
    }


    public static void main(String[] args) {
        MyArrayList<Integer>  list =  new MyArrayList<Integer>();
        for(int i = 0 ; i < 5; i++){
            list.add(i);
            System.out.println(list);
        }
        list.remove(1);
        System.out.println(list);
        list.add(5);
        System.out.println(list);
    }

}
```

5.11.2 Python

```
from doctest import OutputChecker


class MyArrayList(object):
    def __init__(self) -> None:
        self.size = 0
        self.capacity = 10
        self.theData = [None]*self.capacity

    def __len__(self):
        return self.size

    def insert(self, index: int, item):
        if not isinstance(index, int):
            raise IndexError(index + " is not an integer.")
        if index < 0 or index > self.size:
            raise IndexError("Index " + str(index) +  " is out of range.")
        if self.size ==  self.capacity:
            self.__reallocate()

        for i in range(self.size -1, index -1, -1):
            temp =  self.theData[i]
            self.theData[i+1] = temp


        self.theData[index] =  item
        self.size += 1

    def append(self, item):
        self.insert(self.size,item)

    def __reallocate(self):
        self.capacity = self.capacity * 2
        newData =  [None] * self.capacity
        for index, item in enumerate(self.theData):
            newData[index] = item
        self.theData = newData

    def remove(self, index: int):
        if index < 0 or index >= self.size:
            raise IndexError("Index " + str(index) +  " is out of range.")

        item = self.theData[index]

        for index in range(index+1,self.size):
            self.theData[index -1] = self.theData[index]

        self.size = self.size - 1
        return item
```

```
    """
    def __str__(self): # first attempt
        output =  "["
        for item in self.theData:
            output += str(item) +","
        output = output[:-1] # remove the last comma
        return output + "]"
    """
    def __str__(self): # second attempt
        output =  "["
        #only include indexes from 0 to size-1
        for item in self.theData[:self.size]:
            output += str(item) +","
        output = output[:-1] # remove the last comma
        return output + "]"
    # obviated by dunder method
    def get(self, index):
        if index < 0 or index >= self.size:
            raise IndexError("Index " + str(index) +  " is out of range.")
        return self.theData[index]

    # obviated by dunder method
    def set(self, index, item):
        if index < 0 or index >= self.size:
            raise IndexError("Index " + str(index) +  " is out of range.")
        oldItem = self.theData[index]
        self.theData[index] = item
        return oldItem


    def __getitem__(self, index):
        if index < 0:
            index =  index % self.size  # yes!
            # If you're confused, test modulo on
            # negative numbers in python.
        if index >= self.size:
            raise IndexError("Index " + str(index) +  " is out of range.")
        return self.theData[index]

    def __setitem__(self, index, item):
        if index < 0:
            index =  index % self.size
        if index >= self.size:
            raise IndexError("Index " + str(index) +  " is out of range.")
        oldItem = self.theData[index]
        self.theData[index] = item
        return oldItem

l =  MyArrayList()
for i in range(12):
```

```
    l.append(i)
l.remove(2)
print(l)
```

Chapter 6

Linked Lists

Linked lists , also referred to as reference based lists , are the second type of lists typically seen in applications . To be clear a linked list is a list. That means it could be used anywhere an array list can. So Why do we have two objects that are functionally equivalent , two collections that hold things in order, using indexes? The answer is will see, is because each list is good at the thing the other list is less efficient at.

Array based lists use contiguous blocks of memory, allocated all at once and when then capacity of the list is filled up. Utilizing an array makes these types of lists extremely efficient at retrieving an item from a specific index, but adding items anywhere but the end of the list incurs a $O(n)$ runtime.

Linked Lists can do all the things an Array List can, but the underlying structure is completely different. Each item in the list is stored in an Object called a *Node*. Nodes are created as items are added to list, rather than in advance. This means that are not contiguous, but Rather they are scattered throughout the computer's memory . So how in the world do we keep track of where we've stored all these items ? The solution resembles the scavenger hunt through the computer's memory. Each node Not only the memory location of the item that is being stored, but the memory location of the next node in the list . An example of this code can be found below[1]:

```
// a snippet of the Node Class
// This will live inside the LinkedList class
private static class Node<E> {
        E item;
        Node<E> next;

        public Node(E item) {
                this.item =  item;
        }
}
```

[1]Why is this class private in Java `private`? An inner class (or private class) is a class that lives within another class. We use this for two reasons: Our nodes only exist to build the linked list, so they don't need to have their own class. The Second reason is What about `static class`? This means that we can create nodes without having to make a Linked List first!

Upon first glance, this code may be very confusing. Each node class contains a reference to a node inside of it. This may give the impression that nodes situated one inside another, like one of those Russian nesting matryoshka dolls. However, keep in mind what the node is actually storing is not other objects, but instead memory locations of where to find them. This means that our linked list is more akin to a scavenger hunt where each objective in the hunt contains the instructions on how to find the next objective.

In other words, the item Is the data that is being stored (well actually the memory location, don't forget that) , and next refers to the memory location of the next index in the list. Crash course is an excellent video demonstrating this which you can find here:

6.1 Connecting Nodes into a list.

we keep track of only the first and last item in the list, referred to as the head and the tail .

I will be presenting the directions to building a fully functional singly-linked list and doubly-linked list. These directions will differ from the mechanics of how your programming language of choice implements them, but have the same time complexity for their operations. My implementation is constructed with the goal of making the code easy to understand and the decisions that need to be for adding and removing reflect each other. Finally, my code aims to minimize the number of null-pointer exceptions and their ilk a programmer would make.

The full implementations can be found at the end of the Chapter.

6.2 Building a Singly LinkedList

We open up our linked list with a class declaration. If our language uses generics, we specify it there. I'll be choosing not to inherit from the built-in list so we can focus solely on our own code and no external distractions.

In Java, our code begins like this.

```
public class LinkedList<E> { }
```

In Python

```
class LinkedList(object):
pass
```

6.2.1 The Node

We want the Node class to be a private/internal class, so that the Node we write for a singly linked list and doubly linked list won't get mixed up in our coding environments. This also applies for other data structures that will be using nodes.

```
public class LinkedList<E> {

        private static class Node<E>{
                E item;
```

```
                        Node<E> next;

                        public Node(E item){
                                this.item = item;
                        }
                }
        }
```

```
class LinkedList(object):
        class Node(object):
                def __init__(self, item) -> None:
                self.item = item
                self.next = None

        pass
```

In the Node private/internal/inner class (and only there), the `this` or `self` refers to the **node** rather than the linked list.

6.2.2 Instance Variables and Constructor

Our linked list Linkedlist only needs a few Instance variables in order to Function. We need to keep track of the size; Without it we would have no idea what the valid indices are in the list. We need to keep track of the head so we know where to start our scavenger hunt for any particular index or item we're looking for. Finally we'll keep track of the tail . While keeping track of the tail isn't strictly necessary , keeping track of it means that will be able to add an item to the end of the linked list very efficiently (`O(1)`).

The only job of the constructor is to initialize everything to either zero or null.

Finally, it's probably a good idea to go ahead and write getter method for the size of the list.

```
public class LinkedList<E> {
        private Node<E> head;
        private Node<E> tail;
        private int size;


        public int size(){
                return this.size;
        }
}
```

6.2.3 Adding

Our Linked list has two add methods, just like the array list. The first only takes in an item and adds that item to the end of the linked list . It will do this by calling our second method which takes in an index and an item and inserts that item at that index.[2]

[2]If this sounds familiar, it's because this is precisely what the add method in the arraylist does. Shocking, right?

Let's take a look at our first add[3] method:

```
public boolean add(E item){
        this.add(this.size, item);
        return true;
}
```

```
def add(self, item):
        self.add(self.size, item)
        return True
```

Simple enough! But what about that second add method? When we do any kind of operation on a linked list, we need to think about how instance variables in a linked list will be altered. Fortunately, we only have three instance variables: `size`, `head`, and `tail`. When adding to a linked list, the size will always be altered as long as the index is valid. Our list's `head` will only be altered when we add an item to the beginning of the list and our `tail` will only be altered when we add to the end of the list. If the list is empty , then the node for that added item becomes both the head and the tail.

We can simplify our job by breaking the add method into five separate cases:

1. The index that we want to add to is out of bounds.
2. We are adding an item to a list that is completely empty. This is going to change the head and tail the list from nolta something.
3. We are adding an item to index 0, which is going to change the head of the list.
4. We are going to add an item to the end of the list, which means that we are going to change what the tail is.
5. We are adding to some other index in the list , which means that we don't have to bother changing the head or the tail.

Let's start with the first case.

Checking the index is in or out of bounds

Since we passed the check above , we should take a moment before we add an item to address things that need to happen no matter what for Every add condition . Specifically, we need to have a node to hold the item we are adding , and we want to go ahead and increment the size of the list At the end of the method so we don't forget about it.

I will be calling the node that holds the item we are inserting into the list `adding`, As calling it node would be extremely confusing, since we are dealing with so many nodes and other variables like next that are also four letters long.

Here's what our changes look like.

[3] As with the arraylist , the add method returns a boolean to signify that we were successfully able to add it to the list . This will always be true, but we do this because Java expects this for collections, as explained in arraylists

```
public void add(int index, E item) {
        // Scenario 1: index is out of bound
        if(index < 0 || index > size ) {  //O(1)
                throw new IndexOutOfBoundsException(index + " is out of bounds");
        }

        Node<E> adding = new Node<E>(item);
        /* the rest of our code*/
        size++;
```

Adding to an Empty List

Now let's consider Adding to an empty list. An empty list means the size is 0. If that's the case, we are going to make Adding the new head of the list, As well as the new tail. Just like if you are the only person in line at checkout you are both the first person and the last person in line , this node will also be the first node and the last node in the list , which is why it Will be both the head and tail of the list (at least until we add another item).

```
        // Scenario 2: adding to an initially empty list
        if(size == 0) {
                head = adding;
                tail = adding;
        }
```

Adding an item to the beginning of the list

Adding an item to the beginning of the list means that the node containing it becomes the new head of the list. We do this bye attaching Adding to the list, Then informing the list adding is the new head .We do this by setting adding's .next Two point to the current head of the list, then setting The list had to be the node we added.

```
        // Scenario 3: adding a new head
        else if(index == 0) {//(1)
                adding.next = head;
                head = adding;
        }
```

Here, we introduce one of the most important rules we need to follow when working with a linked list : when we are adding an item to the linked list attached the list first , then update the rest of the list to accommodate the new reality.

Failing to do this can have catastrophic results. Consider below Where we set Adding as new head first

```
        // Mistakes were made
        else if(index == 0) {
                head = adding;  // oops
                adding.next = head;
        }
```

Note that the number of operations we do here Is always the same no matter how big the list is! This means that adding to the head is a constant time operation.

Adding an item to the end of the list

```
// Scenario 4: adding a new tail
else if(index == size ){
        tail.next = adding;
        tail = adding;
}
```

Sidebar: Getting a Node at a Specific Index

```
private Node<E> getNode(int index){  //O(n)
        Node<E> current = head;
        for (int i = 0; i < index; i++) {
                current = current.next;
        }
        return current;
}
```

Inserting an item into a specific index

```
// Scenario 5: everything else
else {
        Node<E> before =  getNode(index -1);  //O(n)
        adding.next = before.next;
        before.next = adding;
}
```

The end result

```
public void add(int index, E item) {
        // Scenario 1: index is out of bound
        if(index < 0 || index > size ) {  //O(1)
                throw new IndexOutOfBoundsException("Not a valid index :(");
        }

        Node<E> adding = new Node<E>(item);

        // Scenario 2: adding to an initially empty list
        if(size == 0) {
                head = adding;
                tail = adding;
        }
        // Scenario 3: adding a new head
        else if(index == 0) {  //    O(1)
                adding.next = head;
                head = adding;
        }
        // Scenario 4: adding a new tail
```

```
        else if(index == size ){
                tail.next = adding;
                tail = adding;
        }
        // Scenario 5: everything else
        else {
                Node<E> before =  getNode(index -1);  //O(n)
                adding.next = before.next;
                before.next = adding;
        }


        size++;
}
```

6.3 Get and Set

Before we got onto our remove method, let's take a look at `get` and `set` very briefly.

6.3.1 Get

Just like with an ArrayList, the get method returns the item and the specified index. However, since we can't go directly to a specific index like we can with an array or ArrayList, we need to iterate thru the `.next` links until we get to the appropriate node. Fortunately, we can just use our `getNode` function that we created when we were writing `add`.

```
public E get(int index) {
        if(index < 0 || index >= size ) {
                throw new IndexOutOfBoundsException(index + " is out of bounds");
        }
        return getNode(index).item;
}
```

6.3.2 Set

Set operates very similar to get. Remember, set also returns the item that is already at the specified index, essentially replacing it.

```
public E set(int index, E item) {
        if(index < 0 || index >= size ) {  //O(1)
                throw new IndexOutOfBoundsException(index + " is out of bounds");
        }
        Node<E> node = getNode(index);
        E toReturn = node.item;
        node.item = item;

        return toReturn;
}
```

6.4 Remove

6.5 Analysis

Array lists and linked lists are both extremely powerful objects that fulfill the same purpose, but in radically different ways.

6.5.1 Some Algorithms Play Better

Linked Lists are more efficient for algorithms that require a list to be split, such as Merge sort, or when items are constantly being moved from the front to the back. Linked Lists are also extremely efficient with certain card-like operations, like cutting a deck (eg moving a contiguous group of items starting at index zero of a list to the rear of the list)

However, if your algorithm constantly needs to seek the midpoints between two indices in the list, Arraylists are extremely efficient whilst linked lists suffer with their operations.

6.6 Potential Project/Practice/Labs

6.7 Source Code

```
from typing import Generic, TypeVar


E =  TypeVar('E')

class LinkedList(Generic[E]):

    class Node(Generic[E]):
        def __init__(self, item: E) -> None:
            self.item = item
            self.next = None

    def __init__(self) -> None:
        self.head = None
        self.tail = None
        self.size = 0

    def __len__(self) -> int:
        return self.size

    def getNode(self, index: int) -> Node:
        current = self.head
        for i in range(index):
            current = current.next
        return current

    def add(self, item: E) -> bool:
```

```
        self.add(self.size,item)
        return True

    def add(self, index: int, item: E) -> None:
        if(index < 0 or index > self.size):
            raise IndexError("Invalid add at index " + str(index) +" with item" + str(item) +".

        adding = self.Node(item)
        if(self.size == 0):
            self.head = adding
            self.tail = adding
        elif(index == 0):
            adding.next = self.head
            self.head = adding
        elif(index == self.size):
            self.tail.next = adding
            self.tail = adding
        else:
            before = self.getNode(index - 1)
            adding.next = before.next
            before.next = adding

        self.size += 1

    def remove(self, index: int) -> E:
        if(index < 0 or index >= self.size):
            raise Exception("Invalid remove at index " + str(index) +".")

        toReturn = None
        if self.size == 1:
            toReturn =  self.head.item
            self.head = None
            self.tail = None
        elif index == 0:
            toReturn = self.head.item
            self.head = self.head.next
        elif index == self.size -1:
            toReturn = self.tail.item
            self.tail = self.getNode(index - 1)
            self.tail.next =  None
        else:
            before =  self.getNode(index - 1)
            toReturn = before.next.item
            before.next = before.next.next
        self.size -= 1
        return toReturn

    def get(self, index: int) -> E:
        return self.getNode(index).item

    def set(self, index: int, item: E) -> E:
```

```
        node = self.getNode(index)
        oldItem = node.item
        node.item = item
        return oldItem

    def __str__(self) -> str:
        output = ""
        current = self.head
        while current != None:
            output += str(current.item) + "->"
            current = current.next
        return output[:-2]


l = LinkedList()
l.add(3)
l.add(5)
l.add(142)
```

Chapter 7

Stacks

Our next data structure is the Stack. The stack may seem unnecessary as a data structure after we introduce its features. After all, can't a list do all the things that a stack can do and more?

Working with the limited operations of a allows us to approach problems with a different mindset.

7.1 Stack Operations

The stack operations are limited and simple.

Push Put an item on the top of the stack.

Pop Remove the item from the top of the stack and return it. The item that was underneath the top of the stack becomes the new top.

Peek Return the top of the stack, without removing it.

That's it. That's all there is. It is refreshingly simple. There will usually be additional functions, such as one to check if the stack is empty or a function to get the number of items stored in the stack, but `push`, `pop`, and `peek` are the important ones.

The common metaphor used for this is a stack of pancakes (Figure 7.1). You wouldn't remove a pancake from the bottom of the stack or the middle — that would know over the whole stack! Instead, you move would only want to add or remove pancakes from the top of the stack.[1]

7.2 Building a Stack

We will be building a stack as a reference-based structure in this book. This is so we can get a bit more practice with manipulating nodes.

[1] I find the metaphor silly as that implies there's a situation I would willingly remove pancakes from my stack.

Figure 7.1: Delicious, AI-generated pancakes

Listing (Java) 7.1: The Stack (Java)

```
public class Stack<E> {
    private Node<E> top;


    private static class Node<E>{
        E item;
        Node<E> next;
        public Node(E item) {
            this.item = item;
        }
    }

    public boolean isEmpty(){
        return top == null;
    }

    public E peek() {
        return top.item;
    }

    public E pop() {
        E toReturn =  top.item;
        top = top.next;
        return toReturn;
    }

    public void push(E item){
        Node<E> newTop = new Node<E>(item);
        newTop.next = top;
        top = newTop;
    }


}
```

Listing (Python) 7.2: The Stack

```
class Stack:
    class Node:
        def __init__(self, item):
            self.item = item
            self.next = None

    def __init__(self):
        self.top = None

    def isEmpty(self):
        return self.top is None

    def peek(self):
        return self.top.item

    def pop(self):
        toReturn = self.top.item
        self.top = self.top.next
        return toReturn

    def push(self, item):
        newTop = self.Node(item)
        newTop.next = self.top
        self.top = newTop
```

7.3 Built-in Stacks

Our programming languages have functionality for Stacks built into them, although it is different than our pedagogical model.

7.3.1 The Stack - Java

The built in `Stack` for Java uses an array-based[2] implementation, rather than a reference based implementation, like above. It uses all the conventional Stack method names.

If you are going to use a `Stack` in production, you should instead use a `Deque`. See Chapter **??** for details. Conclusion, use `Stack` in this course, but be prepared to use `Deque` outside the course.

7.3.2 The List - Python's Stack

Python has no separate built-in stack. Rather, we instead use a Rather, it uses the List that we are already familiar with to emulate a stack[3] and operates on

[2]Stack is a subclass of the `Vector` class, itself a subclass of the `AbstractList` abstract class. The `Vector` is extremely similar to `ArrayList` but older. You should not use a `Vector` unless you have an extremely specific reason; I have never had a reason.

[3]And the Queue, as we will see in Chapter 8

the last (right-most) index of the list.

If you want to use a Python list as a stack, merely restrict yourself to using the `append(item)` function in place of `push`. Python lists have a `pop` method; when called without any argument[4], it removes and returns the last element in the list. Use `stackname[-1]` to `peek` at the top of the stack.

7.4 Why?

Why use a stack over a much more powerful data structure? Using stacks (and queues) help focus on seeing if there's a particular strategy for solving a problem. With stacks, that strategy is typically backtracking. Furthermore, limiting the operations of storing and retrieving data to operating on only the front/top of a list-like structure means that we can ensure all storage and retrieval operations run in $O(1)$ time.

7.5 Mazes - Stacks and Backtracking

If you haven't ever done a hedge maze, you should try it out. They are pretty fun in my opinion and certainly doing at least once. That said, I would venture most people playing in a maze of some sort meander through with a vague strategy, picking a direction they hope will get them closer to the goal.

Let me teach you two such strategies for when you get stuck in a maze. The first strategy is the "hand on wall" rule or "right hand" rule. It requires no preparation and works on almost every maze. Simply take your right hand and lay it upon the wall of the corridor you are in. Move forward and when you come to a turn, travel so that you never lift your hand off the wall. So long as the entrance and exit are on the same wall (which they almost certainly are), you'll eventually make your way out.

The second strategy is backtracking[5] and works on any maze you are likely to encounter.[6] Let's explore this using an example from Greek Mythology: the Labyrinth.

7.5.1 The Labyrinth

Once upon a time, King Minos, child of Zeus and big jerk of a demigod, really messed up and angered Poseidon. Minos's wife then gave birth to the Minotaur, a half-man/half-bull. Minos had the inventor Daedalus build the Labyrinth for him to keep the Minotaur in. The Labyrinth was a giant maze, and Minos, being a big jerk, frequently tossed Athenians into the Labyrinth to feed the Minotaur.

This continued until Theseus, son of Poseidon, put a stop to it by navigating the Labyrinth and slaying the Minotaur. Theseus managed to pull this off with the help of Ariadne, Minos's daughter and noted not big jerk. Ariadne provided Theseus with a ball of thread, which allowed him to navigate the Labyrinth.

[4] When provided an index, `pop` removes and returns the item at that index. Python uses `pop` to remove at an index, whereas `remove` is used to remove and return the first occurrence a specified item.

[5] Technically this is going to look a lot like Depth First Search, but this is a very specific variation of it.

[6] The exception is mazes that change their configuration while you are in them.

Minos eventually died in a bizarre series of events involving him being a stalker, a seashell, and even more thread.

It's the thread here that we are concerned with. See, the Labyrinth is classically understood to have a ton of twists and turns and easy to get lost in. Let's put ourselves in the shoes of Theseus for a second and think about how we can use that thread to navigate this maze.

Since we're pretending we're a demigod protagonist of a Greek myth, we might as well pretend that Ariadne's thread is magic too. We won't run out of thread and it can't be cut by the Minotaur or otherwise tangled. Let's also assume we're in one of the versions of the myth where we have a sword. We will let the thread unroll along the ground as a we traverse the maze. When we come to a crossroads or some other choice of passages, we will travel down an unexplored passage.

It's possible we hit a dead end in one of two ways. The first way we hit a dead end is that the corridor ends; a literal dead end. The second way we can hit a dead end is by coming to a crossroads and there are no unexplored passages. In either case, we can backtrack.

To backtrack, we turn around and follow the thread until we find an unexplored passage. We wind the thread back up and mark the floor or walls with the sword to let our future selves know this was a dead end [7]. We know we have found an unexplored passage when find a passage with no thread or sword marks. This will allow us to navigate the entire maze and ensure we only ever traverse a passage twice: once while exploring and once while backtracking.

Abstracting this out, each corridor (or cell of the maze when we get programming) has three states:

Unexplored This is a part of the maze we haven't traveled down yet.

Visited This would be part of the maze we have traversed.

Backtracked This is a part of the maze we have traversed *twice*, the second time being us reversing our trail.

We can use a Stack in place of our magic thread, using the `push` operation on a location to say we have traveled to this location, with the top of the Stack holding our current location. This is analogous to unrolling the thread. If we need to backtrack, we `pop()` and go to the location that's now at the top of the task. This represents the act of us rolling the thread back

Rather than awesome sword, we will use much more mundane `booleans` or `enums` or `colors` to ensure we don't revisit an already explored or backtracked corridor. Which we use depends on our implementation.

This gives us our algorithm.

```
Given: a maze represented by a 2D array of cells
Cell: represents a unit of the maze.
    Has a variable for color or exploration status.
```

[7] If we don't have a sword, we don't wind the thread back up, because if we did, we'd end up essentially erasing markers we have that a passage has been explored. Instead, we will leave a second line of thread along the ground on passages we backtrack upon. We have the sword for pedagogical reasons, a sentence I never thought I would write, but always hoped that I would.

```
    Also has variables to represent walls or lack thereof
    in each of the cardinal directions

Algorithm:
push start position on top of stack
while maze exploration is not done and and stack isn't empty
    peek at the stack to get our current position
    if we can go north and haven't visited there yet
        push the location to the north on the stack
        mark the current location as visited
    else if we can go south...
    repeat for east and west
    else
    we can't go anywhere so we are at a dead end
        mark current as a dead end
        pop off the stack
```

We can assume that maze exploration is done if we find the exit. If the stack is empty, we have come back to the beginning of the maze. The last note I'll make before our next topic is possibly the most intriguing. With a bit of creativity and some minor tweaks, we can take this algorithm and modify it to *generate* mazes rather than solve them. Generating mazes is it's own fun subgenre, as each strategy and algorithm for creating mazes creates mazes with different biases.

7.6 Parenthesis Matching

A classic stack problem is to write a program that checks to see if a given string has balanced parenthesis. As a student, you encounter this problem in three places: a data structures class like this, an interview question, or Computer Automata class.[8] The question we're trying to answer is something like "Does the string `((A + f(x[0])) + B` have balanced parentheses?" We humans can easily take a look and say no, `((A + f(x[0])) + B` doesn't balanced parentheses; it's missing a closing parenthesis. But how did we do that? Codifying that is how we solve this problem.

Now if it was a matter counting the number of opening and closing parenthesis, this would be an easy problem. But in asking if the parenthesis are balanced, we're essentially asking if they match in a way that makes mathematical sense: everything is nested correctly; nothing closed before it opens. Simply counting the correct number of opening and closing parenthesis would fail on `a)(` and `fx)())`.

Instead, we can use a specific tool to handle this problem. If you guessed it is the stack, excellent work. You must have caught on to the numerous hints, such as this being in the chapter about stacks, or the starting sentence talking about how this is a classic stack problem.

We parse thru the string and skip anything not a parenthesis or bracket or the like. When we see an open parenthesis/bracket, we push it onto the stack.

[8] This problem is used to show the limits of Discrete Finite Automata/Regular Languages and introduce Stack Machines/Context Free Grammars

When we see a closing brace, we pop from the stack and compare. If we have a match, no issues. But if the type of brace or bracket or parenthesis doesn't match or there was nothing to pop off, return false.

Listing (Java) 7.3: Parenthesis Matching in Java

```java
public static boolean isBalanced(String expression) {
    Stack<Character> stack = new Stack<>();
    for (Character c : expression.toCharArray()) {
        if(c == '(' || c=='[' || c == '{' ) {
                stack.push(c);

            } else if( c== ')' || c== ']' || c == '}') {
            if(stack.isEmpty()){
                return false;
            }
            char opener = stack.pop();
            if( !((opener=='(' && c==')') || (opener=='['
            ↪  && c==']') || (opener=='{' && c=='}'))){
                return false;
            }

        }

    }
    return stack.isEmpty();
}
```

Listing (Python) 7.4: Parenthesis Matching in Python

```python
def isBalanced(expression):
    stack = []
    for c in expression:
        if c in "([{":
            stack.append(c)
        elif c in ")]}":
            if not stack:
                return False
    opener = stack.pop()
    if not ((opener == '(' and c == ')') or
            (opener == '[' and c == ']') or
            (opener == '{' and c == '}')):
        return False
    return not stack
```

Chapter 8

Queues

A Queue (pronounced by saying the first letter and ignoring all the others) is a data structure which emulates the real word functionality of standing in a line (or queue, for those from Commonwealth nations). In a Queue, items are processed in the order they are inserted into the Queue. So if Alice enters the Queue, followed by Bob, then followed by Carla, Alice would be up first to leave the Queue, then Bob, and then Carla. In other words, the item that has been in the queue the longest is at the front of the Queue and is the next to be processed.

We often refer to a Stack as the LIFO - Last In, First Out - data structure, while the Queue serves as a FIFO - First In, First Out - Data Structure.[1] The use cases for Queues are fairly obvious.

8.1 Queue Operations

The queue operations are also simple.

Enqueue Put an item at the back of the Queue.

Dequeue Remove and return the item at the front of the Queue. The next item becomes the new front.

Peek Return the front of the Queue, without removing it.

After lists and stacks, this should pose no challenge.

8.2 Reference Based Implementation

Much like the stack in Chapter 7, we can create a queue as an extremely simplified linked list.

[1] There's gotta be a joke for GIGO - Garbage in Garbage out, to put here.

Listing (Java) 8.1: A Reference based Queue

```
public class MyQueue<E> {
    // A pedagogical queue
    private Node<E> back;
    private Node<E> front;

    private static class Node<E>{
        E item;
        Node<E> next;
        public Node(E item) {
            this.item = item;
        }
    }

    public void enqueue(E item){
        Node<E> newBack =  new Node<E>(item);
        back.next = newBack;
        back =  newBack;
    }

    public E dequeue() {
        E toReturn =  front.item;
        front = front.next;
        return toReturn;
    }

    public E peek() {
        return front.item;
    }
}
```

Listing (Python) 8.2: A Reference based Queue

```
class Stack:
    class Node:
        def __init__(self, item):
            self.item = item
            self.next = None

    def __init__(self):
        self.top = None

    def isEmpty(self):
        return self.top is None

    def peek(self):
        return self.top.item

    def pop(self):
        toReturn = self.top.item
        self.top = self.top.next
        return toReturn

    def push(self, item):
        newTop = self.Node(item)
        newTop.next = self.top
        self.top = newTop
```

8.3 Built-in Queues

8.3.1 Java's Implementation

The `Queue` - Use on Exams and Psuedocode

In Java, a `Queue` is an Interface with the following methods

`offer(item)` - Java's enqueue.

`poll()` - Java's dequeue.

`peek()` - Java's peek. A peak name.

In the case of the method call failing, the method will return `false` or `null` as applicable. The Queue interface also provides a version of each the methods that do the exact same thing, but throw an exception instead. These methods are `add(item)`, `remove()`, and `element()` respectively.

The `LinkedList` class implements the `Queue` interface, so the most straightforward way to use a queue in Java is to do the following:

Figure 8.1: You, after learning about how to do stacks and queues.

Listing (Java) 8.3: Q

```
ueue<E> q = new LinkedList<>();
q.offer(item); // to enqueue
q.offer(item2); // to enqueue
q.poll() // removes and returns item1
q.peek() // item2 is now the head
```

The `Deque`

The `LinkedList` class also implements the `Deque` interface. A `Deque` is a *double-ended queue*, which means adds and removes happen at either end of the data structure. It's fairly straightforward to see why a `LinkedList` is the perfect class to implement this. Using a `Queue` is perfectly acceptable when you know that all adds will be at the end and removes at the front.

Since a `Deque` is double ended, you can use a `Deque` as a stack. This is actually the recommended way to create a stack in Java as according to the JavaDoc:

> Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy `Stack` class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.

However, `Deque` uses a different naming system for its operations, rather that names like `push` or `pop`. A `Deque` will use `addFirst` instead of `push`, `removeFirst` instead of a `pop`, and `peekFirst` instead of `peek`.

Listing (Java) 8.4: Example `Deque` Usage

```java
Deque<E> stack = new LinkedList<>();
stack.addFirst(item1); push
stack.addFirst(item2); push
stack.removeFirst(); // pop item2
stack.peek(); // will return item1
```

Conclusion, use `Stack` in this course, but be prepared to use `Deque` outside the course. `Queue` or `Deque` for a queue is fine.

8.3.2 Python's Implementation

The List as a Queue

This is your Queue: `[]`. Exciting and super foreign, yes? We can manipulate the list to use it as a Queue like so:

Listing (Python) 8.5: This is bad

```python
q = []
q.append('a')
q.append('b')
q.append('c') # append to enqueue
q.pop(0)      # you can pop index 0 to dequeue
```

However, a keen reader might remember what we know about python's lists and realize that while the enqueue is fast at $O(1)$ time, the dequeue will be $O(n)$ as all the items need to shift to the left.

In fact, our python documentation explicitly says so:

> It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).
>
> To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

Here's their example on how to use that:

Listing (Python) 8.6: Example deque usage adapted from the python docs

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
queue.append("Graham")          # Graham arrives
queue.popleft()                 # The first to arrive now leaves
                                # Yields 'Eric'
queue.popleft()                 # The second to arrive now leaves
                                # Yields 'John'
print(queue)                    # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

A `Deque` is a double-ended queue, meaning it can be used as either a stack or a queue.

Part III

Recursion

Chapter 9

Recursion

9.1 Introduction

9.1.1 Why?

Much in the same way we use Object Oriented Programming as a tool to organize our thoughts about how to design large programs, programmers can use recursion to craft elegant and efficient solutions. Once you get a hang of recursion, it's a really easy way create solutions. I often refer to it as a way to be lazy at programming, with my recursive problem solving typically going like this:

- I am at some amorphous spot in the puzzle or problem I am solving.
- This problem is too big to solve in one go.
- Let's just write code that solves only this specific part of the problem.
- Now that I have the solution to this portion, since I'm lazy, I'll just call a magic method that solves the rest of the problem starting at the point immediately after what I just solves.
- It turns out the magic method is what I just wrote.

Confused? That's fine. It often takes a few attempts to get a handle on recursion. It should start to make sense with some examples.

9.2 Recursive Mathematics

We'll start our discussion with some mathematical examples that you might already be familiar with.

9.2.1 Factorial

The factorial function is hopefully something you have seen before. The function, if not the name, has been know for thousands of years. Here it is in Sefer Yetzerah (4:12)[7] [3], the oldest book of Jewish Mysticism.

שבע כפולות כיצד צרפן. שתי אבנים בונות שני בתים. שלש בונות ששה בתים. ארבע בונות ארבעה ועשרים בתים. חמש בונות מאה ועשרים בתים. שש בונות שבע מאות ועשרים בתים. שבע בונות חמשת אלפים וארבעים בתים. מכאן ואילך צא וחשוב מה שאין הפה יכול לדבר ואין האוזן יכולה לשמוע.

Seven doubles - how are they combined? Two "stones" produce two houses; three form six; four form twenty-four; five form one hundred and twenty; six form seven hundred and twenty; seven form five thousand and forty; and beyond this their numbers increase so that the mouth can hardly utter them, nor the ear hear the number of them.

Mathematically, we use the ! symbol for factorial and define:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n$$

In other words, $n!$ is the product of all the numbers from 1 to n. Thus,

$$1! = 1$$
$$2! = 2$$
$$3! = 6$$
$$4! = 24$$
$$5! = 120$$
$$6! = 720$$
$$7! = 5040$$

0! defined as 1, as we are multiplying no numbers together and the multiplicative identity is 1. Less formally, if you do a running sum, you start at zero, but for a running product, you start with 1, since if you started your running product with zero, you'd get zero.

We can write an iterative implementation of this fairly easily.

Listing (Java) 9.1: Factorial - Iterative

```
public static long factorialIter(int n) {
    long total = 1;
    for(int i = 1; i <= n; i++) {
        total =  total * i;
    }
    return total;
}
```

Notice that I use `long` in Listing 9.1. The total gets very very big, very very fast. Or as Sefer Yetzerah put it: "their numbers increase so that the mouth can hardly utter them, nor the ear hear the number of them."

Now, let's play around with the equation a bit. It's fairly trivial to see in the calculations above that we can get the next value factorial value by multplying by the next integer, e.g. we can go from 2! to 3! by muliplying 2! by 3.

$$1! = 1 \cdot 0! = 1$$
$$2! = 2 \cdot 1! = 2$$
$$3! = 3 \cdot 2! = 6$$
$$4! = 4 \cdot 3! = 24$$
$$5! = 5 \cdot 4! = 120$$
$$6! = 6 \cdot 5! = 720$$
$$7! = 7 \cdot 6! = 5040$$

Going the other direction, we can say that some $n!$ can be figured out by calculating $(n-1)!$ and multiplying by n.

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n \\ &= n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1 \\ &= n \cdot (n-1)! \end{aligned} \tag{9.1}$$

We call this function, where a function is calculated by solving the same function on a (usually) smaller value, a **recursive** function. Let's implement it and take a look.

Listing (Java) 9.2: Factorial - Recursive

```java
public static long factorial(int n) {
    if(n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Listing (Python) 9.3: Factorial - Recursive

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

This probably makes some sense because you were just looking at the math equation, but this might also seem like magic or weird or, worst of all, weird magic. In fact it's quite possible that you've accidentally created something resembling an infinite loop before by having a function or method call itself. So why does it work here?

A recursive function requires two parts in order to work: a base case and a recursive case. The base case is the foundation of our recursive problem. It is where we have a defined solution for some value. In the factorial, this is the line that checks if $n == 0$ in our code, or just defining $0! = 1$ in the mathematics. I look at the base case as the point where we can answer the question reflexively and without much thought.

The recursive case is where we solve our problem by solving a simpler subproblem. In our code, we So in our code, we look at solving `factorial(n)`, decide that's way too much work and decide to solve `factorial(n-1)` and multiply that by `n`. Solving `factorial(n-1)` presents us with the same challenge, so we call `factorial(n-2)` to multiply that against `(n-1)`. Solving `factorial(n-2)` presents us with the same challenge, so we call `factorial(n-3)` to multiply that against `(n-2)`…

This continues until we call `factorial(1)`, which calls `factorial(0)`, the base case, which finally gives us 1.

`facorial(1)` takes that 1 and returns `1 * 1`. Then `factorial(2)` takes the answer from `factorial(1)` and returns `2 * factorial(1)` Then `factorial(3)` takes the answer from `factorial(2)` and returns `3 * factorial(1)` And so on and so forth until `factorial(n)` takes the answer from `factorial(n-1)` and returns `n * factorial(n-1)`

We know this works because for any given non-negative integer[1] n each recursive call on `factorial` is on a smaller and smaller number, making progress to calculating `factorial(0)`. Once we hit `factorial(0)`, the answers start being calculated and trickling up this stack of function calls.

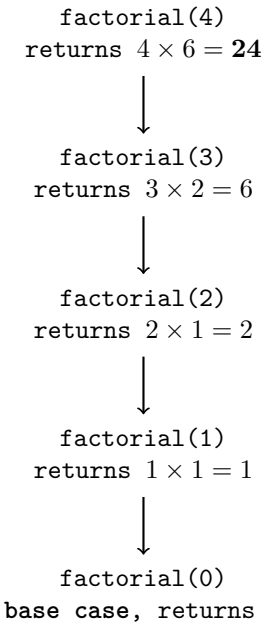


Figure 9.1: The call stack for `factorial(4)`. Each call must wait for the result of the call below it. Once `factorial(0)` returns 1, the results are multiplied back up the stack.

[1]Negative factorials are undefined and I'm ignoring that case in our code. My suggested solution is to either error or document turning something like $(-5)!$ into $-1 \cdot 5!$. It's wrong and will gravely upset the Math department, but might be the desired behavior for your program. But even more important, you should document what you do in weird cases like this!

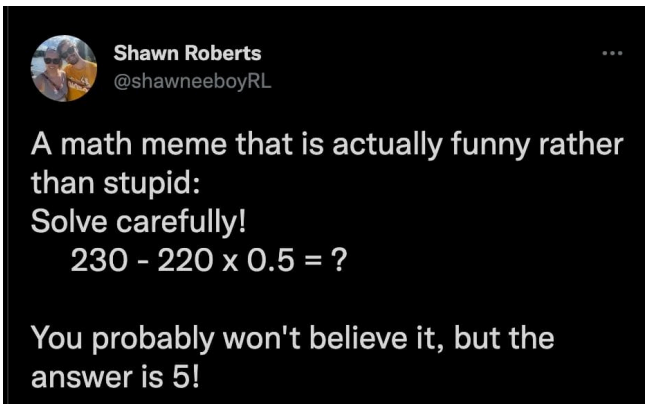


Figure 9.2: Hopefully you get it now.

9.2.2 Recursive Rules

As previously mentioned, all recursive functions:

- Must have one or more base cases where the solution is well defined.
- Must have one or more recursive cases, where the problem is defined by a smaller subproblem of the same type as the original.
- Must ensure the recursive cases make progress towards the defined base case.

You prove a recursive algorithm will solve the problem in question by showing all the above points are true. This is much the same as a proof by induction, just in the opposite direction.

Failure to follow the rules.

If your recursive case fails to make progress towards your base case, then you end up with a special type of infinite loop which is not actually infinite. Every time you make a method call, your computer needs to store where in the code it was and what conditions that were present. The specifics of how and why this is done are outside the scope of the textbook [2], but suffice to say, this information gets stored in a part of the computer memory designated as *the stack*. This stack is named such because it is a `Stack` just like what you have seen in Chapter **??**. Since this stack is living in your memory and your computer probably does not have infinite memory we can run the following program to see what happens when that stack "fills up."

Listing (Java) 9.4: Recursion with no end condition - Java

```
public static void bad(){
    bad();
}
```

[2] maybe

Listing (Python) 9.5: Recursion with no end condition - Python

```
def bad():
    bad()
```

You'll get something along the line of a `Stack Overflow` error or exception, which indicates that your stack in memory has gotten completely used up. This rarely happens in correctly created recursive programs.

9.2.3 Fibonacci

The Fibonacci sequence is the classic introduction to recursive formulas and recursion in programming. I opted for teaching the factorial sequence first due to the complications with runtime a naive implementation has. This might lead to the impression that *all* recursive functions have a terrible runtime. They do not.

History

The Fibonnaci sequence is named after Leonardo Bonacci, also known as Leonardo of Pisa, and also known as Fibonacci. He authored a book in 1202 called *Liber Abaci*, which introduced the western world to calculations using Hindu-Arabic numerals . It also enumerated the Fibonacci sequence, which is why it is named after him[1, 4]. Notice I said *western* world. It is hard to appreciate that humans could not share information in the same way we can today, as well as what can be lost due to damage or merely not writing it down. The Fibonacci sequence had been observed previously by Indian mathematicians such as Gopāla [5] Furthermore, it is completely possible someone had observed this sequence earlier, wrote it down in a text somewhere, and then the text being lost to fire or rot. Backups are important and can mean the difference between having something named after you or not.

Definition

The Fibonacci sequence (sequence A000045)[3] is defined as the sequence of numbers where each number in the sequence is the sum of the previous two numbers. The sequence starts with 0 and 1 and looks something like this:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987...$$

The sequence continues indefinitely. More formally, let F_n be the nth number of the Fibonacci sequence. We define the sequence with:

$$F_0 = 1, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Regardless, the Fibonacci sequence important. It shows up again and again in nature, in science, and in mathematics. The number of petals a flower has tends to be plucked from the Fibonacci sequence [6].

[3] Yes, humans are such nerds that we've created an online library for sequences - OEIS.

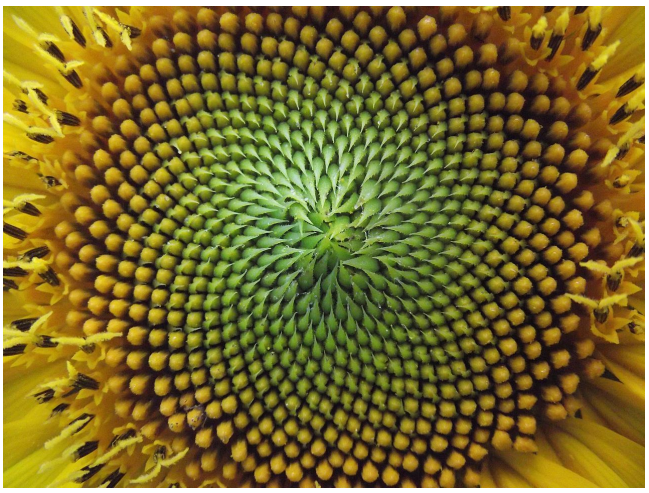


Figure 9.3: The sunflower's fibonacci spiral. Photo by Anna Benczur, CC by-SA 4.0.

Implementation

Implementing this as a recursive function is rather trivial!

Listing (Java) 9.6: Naive Java Implementation

```java
public static long fib(int n){
    if(n == 0 || n == 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Listing (Python) 9.7: Naive Python implementation

```python
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

A Flaw appears in the plan

As it turns out, while this technically works...it's pretty terrible. In short, using recursion, I managed to accidentally[4] write an $O(2^n)$, or exponential time, algorithm. This is very bad. This means increasing n by one *doubles* the runtime of out algorithm! Go ahead and try it for yourself on your computer. You should start seeing some massive slowdowns when computing `fib(n)` somewhere around `n=45`. Notice that each time you increase `n` by one, the amount of time your computer spends working roughly doubles.

[4] All right, I did this totally on purpose.

This is because to solving the current `n` requires solving `fib(n-1)` and `fib(n-2)`. Furthermore, each recursive call is independent from each other; solving `fib(n-1)`

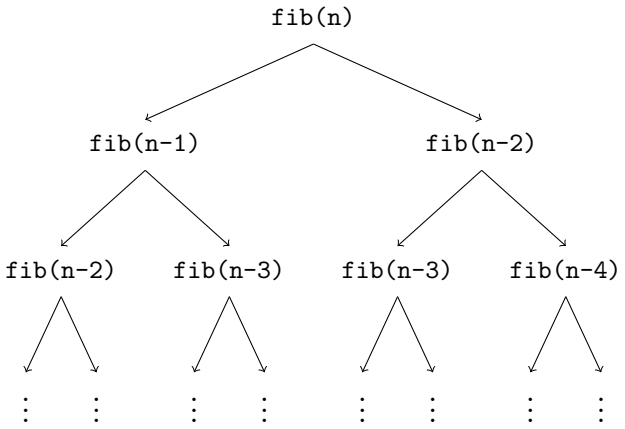


Figure 9.4: Recursive Function Calls for `fib(n)`. Notice that the call to `fib(n-1)` must independently compute `fib(n-2)`, thus duplicating a ton of work.

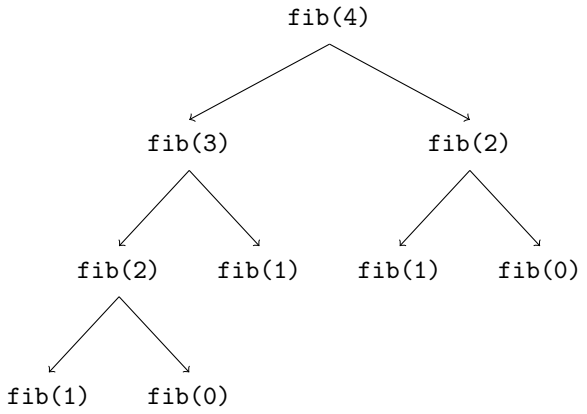


Figure 9.5: Computing `fib(4)`.

Don't let this terrible runtime scare you away from recursion! Recursion can make things quite efficient; this is merely an exception and presented here because Fibonacci is such a classic example we would be remiss to not include it.

Solutions

There's a lot of solutions to make this work. My personal favorite is **memoization**, which simply says "well if the issue is having to redo the work, let's instead store the results of each function call."

Listing (Java) 9.8: An Efficient Recursive Fibonacci Algorithm

```
public static long fib(int n) {
    long[] lookup = new long[n];
    lookup[1] = 1;
    return fib(n , lookup);
}


private static long fib(int n, long[] F) {
    if(n <= 1) { //base case
        return F[n];
    }
    if(F[n-1] == 0) {
        F[n-1] = fib(n-1, F);
    }
    if(F[n-2] == 0) {
        F[n-2] = fib(n-2, F);
    }
    return F[n-1] + F[n-2];
}
```

So here we have a public method that the programmer will use to calculate the `n`th Fibonacci number and a private helper method to do the actual work. The array `F` is an array where we store any previously calculated Fibonacci numbers. The big change from our original solution is now we ask if the `n-1` Fibonacci number has been calculated before. If it has not, calculate it and store it in the array. We do the same for the `n-2` Fibonacci number. The reference to the array is shared between all recursive calls. After the check and the possible calculation is done, the function uses those numbers to calculate `fib(n)`.

Listing (Python) 9.9: An Efficient Recursive Fibonacci Algorithm in Python

```
def fib(n, F = []):
    if len(F) == 0:
        F = [0] * n
    if(n <= 1):
        return n
    if(F[n - 1] == 0):
        F[n - 1] = fib(n - 1,F)
    if(F[n - 2] == 0):
        F[n - 2] = fib(n - 2,F)
    return F[n - 1] + F[n - 2]
```

So here we have a function with a default variable `F` that is initially an empty list. If the program detects `F`'s empty, it is initialized to a list of zeroes. We do this to avoid writing a second function, like we did in the Java example. The list `F` is a list where we store any previously calculated Fibonacci numbers. The big change from our original solution is now we ask if the `n-1` Fibonacci number has been calculated before. If it has not, calculate it and store it in `F`. We do the same for the `n-2` Fibonacci number. The reference to `F` is shared between all recursive calls. After the check and the possible calculation is done, the function uses those numbers to calculate `fib(n)`.

9.3 More Examples

Some of the upcoming examples of the things we are about to see should not be actually used and serve only as examples, like our `printThis` function.

9.3.1 Printing Recursively

Listing (Java) 9.10: Recursive Printing: Java

```
public static void printThis(String s){
    if (s.length() == 0) {
        System.out.println();
    } else {
        System.out.print(s.charAt(0));
        printThis(s.substring(1));
    }
}
```

Listing (Python) 9.11: Recursive Printing: Python

```python
def printThis(s):
    if len(s) == 0:
        print()
    else:
        print(s[0], end='')
        printThis(s[1:])
```

9.4 Arrays with Recursion

9.4.1 Summation of an Array

The ay to think of this is in terms of a base case and a recursive case. The base case is size of one or zero. Either way, the answer is trivially easy to figure out. The recursive case is basically a way of saying adding a bunch of numbers is too hard. I'll just return adding the number at the first index of this section or subsection of the array to whatever the total the rest of the array is. I'll use a magic function to figure it out. it turns out the magic function is actually this one.

9.4.2 Recursive Linear Search

By this point, you know how to iteratively search a list for a specific item. We start at the first item/index and go thru the array one item at a time until we get to the last item or find the item we want. Let's take a look at the same algorithm, just implemented recursively.

Listing (Java) 9.12: Recursive Linear Search - Java

```java
// We return the index we found the item at
// -1 means item is not in the list
public static <E> int search(List<E> list, E target){
    return search(list, target, 0);
}

private static <E> int search(List<E> list, E target, int
↪  index) {
    if(index >= list.size()){
        return -1;
    }
    if(list.get(index).equals(target)){
        return index;
    }
    return search(list,target, index+1);
}
```

Listing (Python) 9.13: Recursive Linear Search - Python

```
def search(theList, target):
    return search(theList, target, 0)

def search(theList, target, index):
    if index >= len(theList):
        return False
    if theList[item] == target:
        return True
    return search(theList, target, index + 1)
```

Again, this is more of a case of pedagogical examples, rather than practical ones. We want to get some practice in before we get to the really interesting recursive problems.

Runtime

The above code has the exact same runtime as doing it iteratively – $O(n)$ in the case of an `ArrayList`. Remember, we don't want to use this for a `LinkedList` due to the $O(n)$ cost the `get` method incurs, which would yield an overall $O(n^2)$ runtime. Use the built-in iterator instead, i.e. use a for each loop.

9.4.3 Binary Search

Binary search is our reason for including Recursion at this location in the textbook. It will be an essential step in building Binary Search Trees.

Objective

Like our recursive linear search, our goal is to search for a particular item in an array or list. Once we find that item, we can either return true or the index we found it at, depending on our implementation. If we fail to find, we return either false or -1 or `null`; again this depends on our implementation.[5].

Assumptions

We will be using an array for Java and `List` for Python. This data structure will be sorted. This is a key assumption; if the array is not sorted, we cannot do a binary search.

Solution

Since our core assumption is that we are using a sorted collection, it makes sense that our algorithm exploits this. Think of a game that you might have played in school as a kid, the "I'm thinking Of a number from one to 100. I'll tell you if it's higher or lower." Now the linear strategy that we went over previously would be the equivalent of asking "Is it one? Oh, it's higher? Is it 2? It's higher? Is it 3? Oh, it's higher?" and so on and so forth Until we hit the number in

[5]Or force a win using *Gifts Ungiven*. Wait, wrong fail to find.

question. A more reasonable strategy would be to pick the number 50 because that number is in the middle of the entire range. Once we know whether the number is higher or lower we have effectively halved our range. This is because if the number is higher than 50 we know that the number cannot be between 1 and 50, inclusive. If it is lower than 50 we know the number cannot be between 50 and 100, inclusive. And if the number is 50 we just simply got lucky. The next step is to choose the number in the middle of our new range so we can do the same halving of our search space.

Let's take this strategy and apply it to an array of sorted numbers, seen in Figure 9.6.

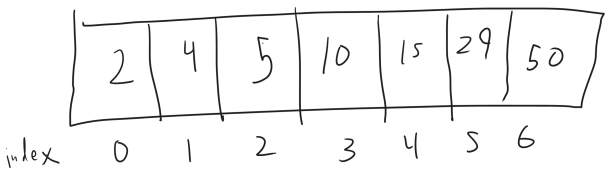


Figure 9.6:

In this example, we want to see query if this array contains the item 5. We start by asking figuring out what the middle index of the array is, since half the items in the array will be to the left and half to the right[6]. The array is 7 items total, so we start at index 3 and compare 5 to the value stored in there (Figure 9.7).

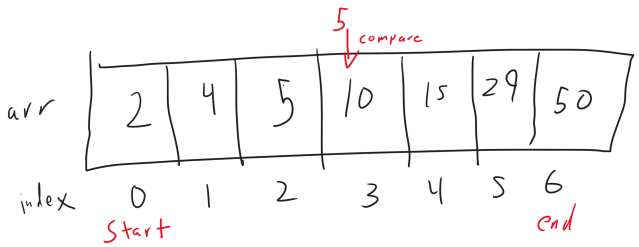


Figure 9.7: The labels `start` and `end` represent the start and end of our search space. This will make more sense as we progress, especially as we start coding.

Since $5 < 10$, we know that if 5 is in the array, it will be found to the left of index 3. We put the end of our search space one index to the left of the middle of our previous search space. Our new range to search is now index 0 thru index 2 (Figure 9.8). We compare 5 to the item in the middle of that range, which is the number 4 at index 1.

$5 > 4$, so if 5 is in the array, it is on the right side of our search space. Our search space contracts to a single item, index 2 (Figure 9.9).

The item at index 2 is the same item we've been looking for, so we have successfully found our item.

[6]Or half stored in lower indices and half stored in higher indices if you prefer.

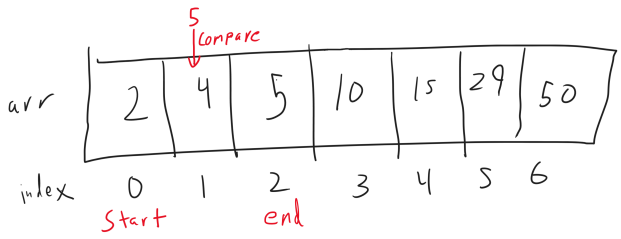


Figure 9.8: The labels `start` and `end` represent the start and end of our search space, which has now shrunk to less than half the original array.

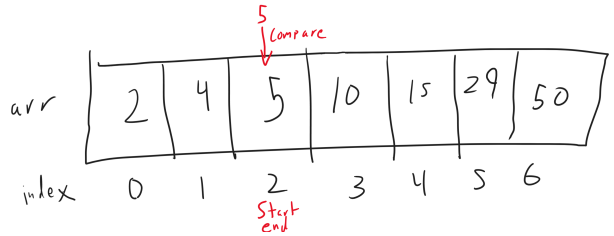


Figure 9.9: The labels `start` and `end` are now on the same item. This means a search space has a size of 1.

Code

Listing (Java) 9.14: Binary Search - Java

```
public static int binarySearch(int[] arr, int target) {
    return binarySearch(arr, target, 0, arr.length-1);
}

private static int binarySearch(int[] arr, int target, int
↪   start, int end) {
    if(start > end) {
        return -1;
    }
    int mid = (start + end) / 2;
    if(target == arr[mid] ) {
        return mid; // item found
    } else if( target < arr[mid]) {
        // search right side
        return binarySearch(arr, target, mid+1, end);
    } else {
        // search left side
        return binarySearch(arr, target, start, mid-1);
    }
}
```

Our outer, wrapper function exists to have a clean function to call. Our helper function does the actual work. If we fail to find our target, we will return -1, which is an invalid index.

Listing (Python) 9.15: Binary Search in Python

```
def binarySearch(arr, target, start = 0, end = 0):
    if len(arr) > 0 and end == 0:
        end = len(arr) - 1
    if start > end:
        return None
    mid = (start + end) // 2
    if arr[mid] == target:
        return mid
    elif target < arr[mid]:
        return binarySearch(arr, target, start, mid - 1)
    else:
        return binarySearch(arr, target, mid + 1, end)
```

Here, we have two base arguments for our initial call, with start and end to be used for a recursive call. The first if statement is to set `end` correctly for our topmost (initial) call, because I'm too lazy to write a second, private helper function. The first base case returns `None` to represent a failure to find.

In our above code, we create two base cases, which are quite simple if you think about it. The first is if our search space is size 0 or invalid. We obviously can't find `target` in the search space if the search spaces doesn't exist. This is the `if(start > end)` clause, since if the start of the search space is to the right of the end of the search space, we don't have a valid space to search anymore. In this scenario, we return a failure state of some sort[7].

The next step is to calculate `mid` which is the index in the middle of the search space. From there, we get our second base case: if the item at index `mid` is the item we are looking for, we're done. Otherwise, we check if `target < arr[mid]`. If this is true, then target must be on the left half of the array. To search that left half for `target`, we call `binarySearch(arr, target, start, mid - 1)`. Why are each of the arguments what they are?

- The first parameter of `binarySearch` is the array or list we are search, so the argument that we pass into our recursive call remains the same.
- The second parameter is the `target` item, which remains constant.
- The third parameter is the beginning index of our search space. When we search to the left side of the search space, we are searching between `start` and `mid`, not including `mid`. Thus the third argument won't change.
- We pass in `mid - 1` as the `end` of the new search space, since that is the right side of the new, smaller search.

Now, if `target < arr[mid]` is false, we search the right side, which means that the fourth argument is `end`, but we change the third argument to `mid + 1`.

[7] -1 in our Java example and `None` in our python example.

Runtime Analysis

Each call of `binarySearch` eliminates either exactly or almost exactly half of the search space if we don't find our `target`. This halving can be mathematically described by the operation $\log_2(n)$, where n is the number of items.[8] Thus, with an array of 256 items, this algorithm would take approximately 8 steps. Doubling the size of the array to 512 items increases the amount of work only by a single step. This yields $O(\log n)$ as the runtime.[9]

Compare that to our linear search, which starts at the beginning and goes through the array one item at a time. That takes $O(n)$ time. In this case, doubling the number of items means doubling the amount of work the algorithm has to do.

It bears emphasizing and repeating: $O(\log n)$ runtime is a major improvement over a linear runtime. Doubling the size of n does practically nothing to change the runtime of `binarySearch`, but would make a linear search take twice as long.

How to not be scared of logarithms

You may have learned that logarithms are the inverse operation to exponentiation. This is an utterly useless definition when programming.

A much more useful way of thinking about logarithms is "how many times can I recursively split something?" For example, $\log_b x$ asks "how many times can I recursively split my x items into b seperate piles?"

A more concrete example: $\log_2 16 = 4$, not because $2^4 = 16$, but because a pile of 16 items can be split in half into two piles of 8, each pile of 8 can be split in half into two piles of 4, the 4's can be split into 2's, the 2's into 1's — four splits total:

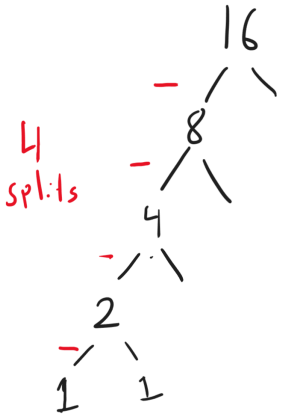


Figure 9.10:

In algorithm analysis, $\log n$ in the time complexity is used to indicate that the search space gets split in half. In the Binary Search algorithm above, we

[8] If the sudden appearance of logarithms risks scaring you off, just keep reading to the next subsubsection. I wrote that special for you.

[9] We drop the base of a logarithm when we use big O notation.

split the our search space in half each step of the way. We start out looking at the middle item and then decide to look at all the items below or all the items above. This reduces the number of items to search among from n to $\frac{n}{2}$. From there we perform the same choices and reduce that $\frac{n}{2}$ to $\frac{n}{4}$, then from $\frac{n}{4}$ to $\frac{n}{8}$ and so on.

Additional Implementation: Java with Lists

Listing (Java) 9.16: Binary Search - Java Lists

```
public static <E extends Comparable<E>> int
↪   binarySearch(List<E> list, E target) {
    return binarySearch(list, target, 0, list.size()-1);
}

public static <E extends Comparable<E>> int
↪   binarySearch(List<E> list, E target, int start, int
↪   end) {
    if(start > end) {
        return -1;
    }
    int mid = (start + end) / 2;
    if(target.compareTo(list.get(mid)) == 0) {
        return mid; // item found
    }
    if(target.compareTo(list.get(mid)) < 0) {
        // search right side
        return binarySearch(list, target, mid+1, end);
    } else {
        // search left side
        return binarySearch(list, target, start, mid-1);
    }
}
```

Performing binary search on a list looks something like this. Recall that `Comparable` is an interface that Java uses to let methods and classes know something can be put in order[a]. This necessarily means that they can be sorted. The generic `<E extends Comparable<E>>` means the the `List` of `E`'s is guaranteed to be made up of items that can be compared to other things of type `E` to see which comes first.

[a]Formally, this is a *total ordering* in fancy math lingo, which means any two items have an established order

9.5 Recursive Backtracking

Recursion really comes in handy when we are trying to solve complex puzzles. One of the most famous examples of this is using recursion to solve the eight

queens problem or Sudoku. Before we get into this, let's establish the rest of the chapter. I'll introduce the eight queen's puzzle. Then, I'll show you the generic recursive backtracking algorithm, explain it, and then show a partial solution to the eight queens puzzle. I'll also go over the Sudoku solver, but will leave that as a potential homework.

The entire reason for this aside is to speak to both the students and instructors who use this book. The eight queens problem is a recursive problem that has been used as a homework problem longer than I've been alive. This means there are a million and a half solutions floating around the internet.

Students: I ask you do not rob yourself of that learning experience and instead strive to follow the text I have here. Go to your teacher or TA or classmate if you get stuck for more than two hours.

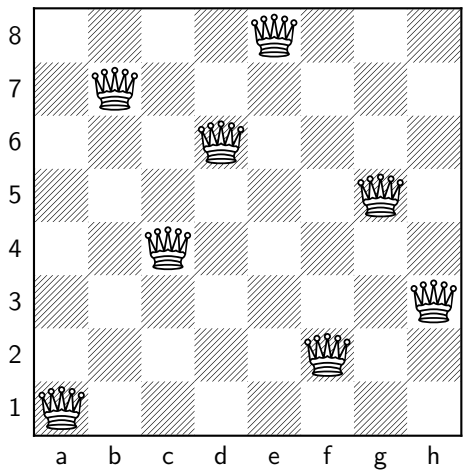
Teachers: Give these exercises, or something similar like the knight's tour, as a homework assignment. These are engaging and worth it. Yes, some students are going to completely ignore the plea of the last paragraph, but they were going to do that no matter what assignment you gave. Don't rob the engaged students of a great learning experience.

9.5.1 The Eight Queens Puzzle

The eight queens puzzle is an old chess puzzle. If you don't know how to play chess, you should change that, but knowing how to play is not necessary to solving the puzzle.

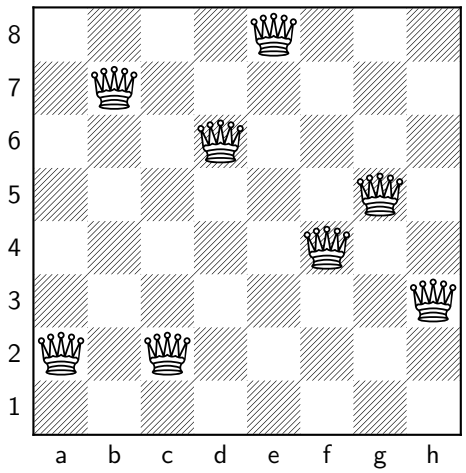
The goal of the puzzle is to place eight queens on a standard 8×8 chessboard. The queens should be placed in such a way that no queen can capture any other queen. In chess, a queen can capture any piece by traveling vertically, horizontally, or diagonally until it crashes into a piece. Or in more programming friendly terms, we need to place 8 queens such that none share a row, column, or diagonal.

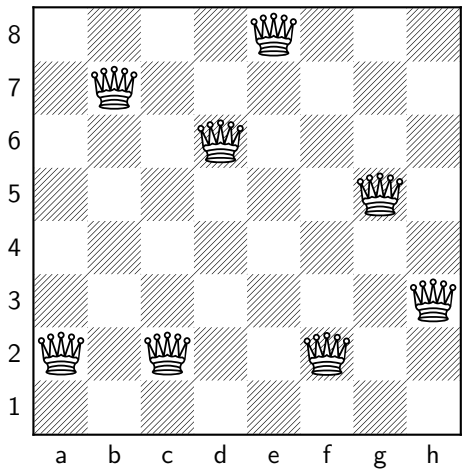
This is one example solution, but it is not the only solution.



Brute Force Solution

A brute force algorithm means we will be checking every single possible state to find a solution. In this case, a brute force solution for the Eight Queens Puzzle would every possible placement of eight queens on a chessboard, such as these two incorrect solutions:

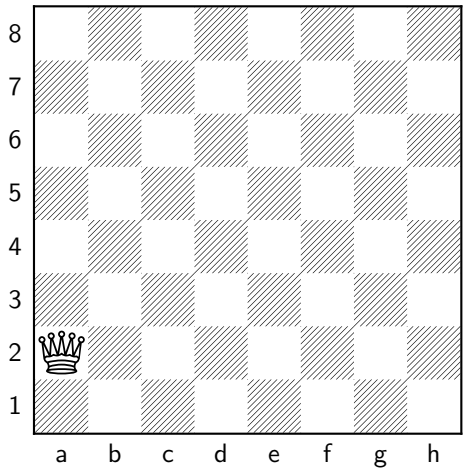


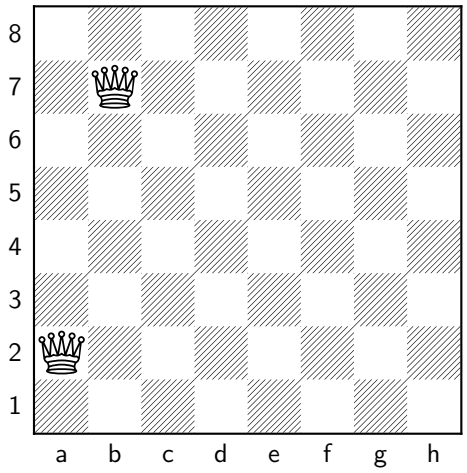


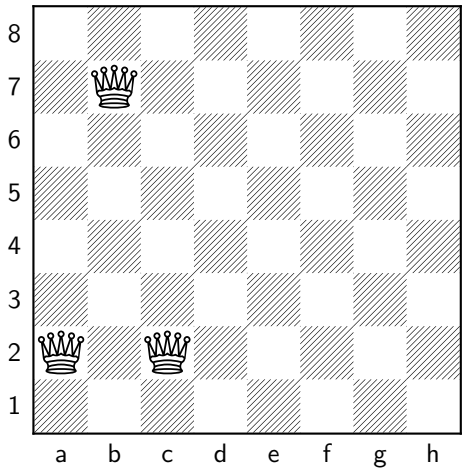
There are a total of $\binom{64}{8} = 4426165368$ possible ways to place 8 queens on a chessboard with 64 spaces. Our computer could sift thru all possible configurations until it finds a solution, and thus, performing a brute force solution will eventually work. It just won't work fast.

Our motive is to do better and apply some more logic to our searches. Take those two incorrect solutions, for example. The first has queens on spaces a2 and c2, which means that it is an incorrect solution. The second example also has queens on a2 and c2, but additionally it has a queen on f2. If we can go about finding a solution more methodically, there is no reason that we should ever check the second example. Once we establish that placing a queen on a2 and c2 doesn't work for a solution, we should never check any solution that

contains a2 and c2 ever again. If fact, let's try and go even further. Assume to generate a solution,we place queens right to left, going one column at a time, like so:







Once we hit this point, there is no need to place any more queens, since we know any of the remaining 5 queens we place down will be in a "dead" branch of the search space. This would allow us to "skip" checking the $\binom{40}{5} = 658008$ solutions that place the five other queens.

The Recursive Backtracking Algorithm

So how do kill a branch of our search space off?

We're going to generalize the puzzle for our next step. Let's introduce you to `solve`, our recursive backtracking algorithm.

```
boolean solve(board,  pos){
```

This solve method will take in two parameters: `board` and `pos`. Board is the puzzle we are trying to solve and all the work we've done on it so far. In eight queens, it will be our chess board and where we've paced the queens. For Sudoku, it would be the grid and all the numbers we have put down (more on that later). For a maze, such as in Chapter 7.5, this would be the configuration of the maze, e.g. the walls and corridors and any marks we have made to avoid searching the same thing twice.

Now, `pos` is our *position* in the puzzle. Let's think of it as the current thing we are trying to solve. For the Eight Queens puzzle, our position the current column we're placing our queen on (more on that later). For Sudoku, we need to find what number goes in this current square. In the maze, we want to select the path leads to the end.

Finally, the `boolean` return value is used to signal whether solved the puzzle or failed.

So now that we are given the two parameters we need, we need to come up with a nice, concise algorithm that solves the entire puzzle. That's a bit of a tall order, but since we are in the Recursion chapter, we should probably think of a recursive solution[10]. That's good, because that will enable us to be *lazy*.

[10]Metagaming is a valuable skill in not just in games, but in school and life. Don't just study the material is for the exam, predict kind of questions your Professor likes to asked based on prior knowledge.

Our recursive solution requires the two elements all recursive solutions need - a base case and a recursive case. We'll also find we need a third case, but more on that in a bit.

```
boolean solve(board,  pos){
        // base case

        // recusive case
}
```

Let's start with the base case. The base case should be the simplest, laziest thing we can think of. For any puzzle, that would be the board already being solved. If our position indicates everything is done, we return true to signal that the maze is solve. This would be having no queens left to place, or no Sudoku squares to fill, or having found the exit.

```
boolean solve(board,  pos){
        // base case
        if(pos indicates puzzle is solved){
                return
        }
        // recusive case
}
```

Easy. Now onto the recursive case. The puzzle is not solved. So let's do the *next laziest* thing - solving our current position, the bare minimum. Now at our position, we have lots of possible choices.

We can think of it as which choice is needed to make to solve the *entire* puzzle from here. Where do we need to put the queen so it will be valid for all the other choices? What number is the correct number to put in this square? What turn in the maze will lead us to the exit? There is no way to know if we're not at the base case, but we're going to pretend that the choice we're making is the correct one for now.

What makes this better than brute forcing a solution is that not every possible choice we can make is valid. How do we figure that out which choice is valid? That depends on the puzzle, so we are going to be lazy and abstract checking if a choice is valid to a `valid` function.

This brings us to the first part of the recursive step. Look at each choice we can make, one at a time. As soon as we find a valid choice, we're going to pick it and assume it's correct.

```
boolean solve(board,  pos){
        // base case
        if(pos indicates puzzle is solved){
                return
        }
        // recusive case
        for each possible choice {
                if(valid(choice)){
                        select and mark that choice;
                        // part two goes here
```

```
                }
        }
}
```

This next part is the trickiest when we first look at it, but once we get the hang of it, writing `valid` will be the hardest part of doing this kind of algorithm.

Now that we've made our presumably correct choice, we are going to be lazy and call a magic `solve` function to solve the rest of the puzzle from the next position. That function will return true or false, depending on whether the magic `solve` function found a solution or not. If it returns true, we will return true to indicate to whatever function called us that the puzzle is solved with the found solution stored in *board*.

If `solve` returns false, that's going to indicate this choice will not lead to a solution, given the current state of the *board*. When that happens, we will need to undo whatever we did to mark our current choice and resume checking the other possible choices to find if one is valid.

```
boolean solve(board, pos) {
        // base case
        if(pos indicates puzzle is solved) {
                return
        }
        // recusive case
        for each possible choice {
                if(valid(choice)) {
                        select and mark that choice;
                        if(solve(board, nextPos) == true){  // recursive case
                                return true;
                        }
                        unmark board at pos if needed, as choice was invalid;
                }
        }
        // But what if there's no valid choice?
}
```

As it happens, the magic function to solve the rest of the puzzle is the one we are currently writing, so we're almost done.

Unlike our previous recursive problems which only had a base case and a recursive case, our solution involves a failure state of sorts. What if we look at all our choices and none of them work? If we hit this case, our program knows it won't be able to find a solution with the current configuration and uses `return false` to inform the function that called it that something in the current configuration needs to change.

This failure case is what makes this a recursive backtracking algorithm.

```
boolean solve(board, pos) {
        // base case
        if(pos indicates puzzle is solved) {
                return
        }
        // recusive case
```

```
        for each possible choice {
                if(valid(choice)) {
                        select and mark that choice;
                        if(solve(board, nextPos) == true){  // recursive case
                                return true;
                        }
                        unmark board at pos if needed, as choice was invalid;
                }
        }
        return false; // backtrack
}
```

As I previously mentioned, while this looks complicated at first, especially the `if(solve(board, nextPos) == true)` once we get used to it, the recursive part is fairly straightforward.

A **key feature** of this is that we always know where we are; we never have to search for the space we are trying to solve - it is the current position.

9.5.2 Recursively Solving the Eight Queens Problem

Now that we have generalized algorithm, let's apply it to the Eight Queens problem. We'll represent the chess board as a 2D array. There's many different valid type we could use, but I'm going to use an array of `String`, using `'Q'` and `'-'` to represent a space with a queen and an empty space respectively. Our position in the puzzle with be the `col` variable, representing the column we are currently working on. Finally, the choice for each column will be what `row` we place the queen on.

Listing (Java) 9.17: Outline of Solution - Java

```java
public static boolean solve(String[][] board, int col){
    if(col == 8) { // use board.length to generalize
        return true;
    }

    for(int row = 0; row < 8; row++) {
        if(valid(choice)){
            place "Q" at row,col
            if(solve(board, pos + 1) == true){
                return true;
            }
            replace "Q" with "-", as choice was invalid
        }
    }
    return false; // backtrack
}
```

The initial call to solve will pass in `0` for `col` to start at the first column. We use `col == 8` as the base case, as a call to `solve(board, 8)` would be trying to check a column that does not exist. The only way for that to happen is from a call on the 8th column (index 7), which means that that last column has found a working solution. If we want to generalize this solution to work on square boards other that a standard 8x8 chess board, we can do that by replacing all the 8's in the code with `board.length`.

Listing (Python) 9.18: Outline of Solution - Java

```python
def solve(board, col):
    if(col == 8): # use len(board) to generalize
        return True

    for row in range(8):
        if valid(choice):
            place "Q" at row,col
            if solve(board, pos + 1) == True:
                return True
            replace "Q" with "-", as choice was invalid
    return False # backtrack
```

The initial call to solve will pass in `0` for `col` to start at the first column. We use `col == 8` as the base case, as a call to `solve(board, 8)` would be trying to check a column that does not exist. The only way for that to happen is from a call on the 8th column (index 7), which means that that last column has found a working solution. If we want to generalize this solution to work on square boards other that a standard 8x8 chess board, we can do that by replacing all the 8's in the code with `len(board)`.

A Place Holder For Validity

Assume for a second that it is possible to not find a solution. Would our program continue forever? We ask this question to make sure the recursion is valid. Let's modify the `valid` function to do nothing but return `false` and examine the result. Whenever we fail to find a solution on a column, `solve` returns `false` to whatever function called it. Thus, if no solution is possible, we will eventually exhaust the search space and return `false` on the entire problem.

Performing the Recursion

Let's go ahead and fill in that code to demonstrate the recursion works. We can do this by setting `valid` function to do nothing but return `true`. Once we do that, there's basically only the most simple of changes to make. Test by calling `solve` by passing in an 8x8 array of `"-"` and `0`.

Listing (Java) 9.19: Solve - Java

```
public static boolean solve(String[][] board, int col){
    if(col == 8) { // use board.length to generalize
        return true;
    }

    for(int row = 0; row < 8; row++) {
        if(valid(board, row, col)){
            board[row][col] = "Q";
            if(solve(board, pos + 1) == true){
                return true;
            }
            board[row][col] = "-";
        }
    }
    return false; // backtrack
}
```

Listing (Python) 9.20: Solve - Python

```
def solve(board, col):
    if(col == 8): # use len(board) to generalize
        return True

    for row in range(8):
        if valid(board, row, col):
            board[row][col] = "Q"
            if solve(board, pos + 1) == True:
                return True
            board[row][col] = "-"
    return False # backtrack
```

The result should be eight queens in the first row and nowhere else, which makes sense; the `valid` function will return `true` no matter what, so the first square in the column is always valid.

Completing `valid` is left as an exercise for the user. Now you might be thinking since Queen moves in eight possible directions, you need to check for conflicts with placing a queen in eight directions too. Fortunately, you're wrong. Firstly, we are only ever placing a single Queen on a column. Since we are trying to figure out which row to place that Queen on for any particular column, we never have to check if the Queen has any pieces above or below it. Second, we are only ever moving from left to right. Whenever we place a Queen, the next column we choose is always to the right. If we don't find a valid queen, the column is cleared before returning `false`. This means we never have to check for Queens to the right of the space we want to place our Queen.

This leaves only three directions to check:

- Directly to the left.
- The upper left diagonal.
- The lower left diagonal.

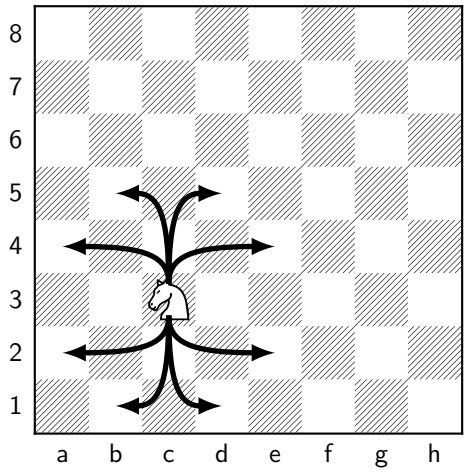
My advice is to work on the first case and test that. If it works, you'll get a line of Queens going from the top left corner to the bottom right.

9.5.3 Additional Problems left to the Reader

Knight's Tour

In the Knight's Tour, we place a Knight on the chess board and move him until he visits each square of the chess board exactly once.[11] A knight moves two squares horizontally or vertically and then one square in the axis it did not move it, creating a sort of "L" shaped (see below). A square counts as visited once the knight lands in it.

[11] Please do not sack Constantinople on your way to the answer.



You may start your knight anywhere you like. Your output should be either the chess board, but with each square marked by a number to designate the order in which the square was visited, or by listing the moves the knight makes. If you can figure out a better way to represent your answer, we are open to that too.

Sudoku

Sudoku (Japanese: 数独 [12] or ナンバープレイス) is a grid based number puzzle, as seen below.

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

The goal is to fill out the puzzle so there are no blank squares. Each square must take a single number from 1 to 9. No number may occur more than once

[12]. 数独 itself is a portmanteau of 数字は独身に限る, roughly "The numbers must occur only once."

in any row, column, or 3x3 box (indicated by the thicker lines). For the sake of simplicity, assume all puzzles are 9x9 and have a unique solution. Thus, the solution to the above puzzle would be:

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

While a human would use logic to find a solution, our `solve` function will be a bit more brutal, plugging in numbers one at a time until we find one that works, backtracking to the previous square when a dead end is reached. Psuedocode would look something like this to begin with:

```
solve(board, row, col) {
        return false; // backtrack
}
```

This indicates that out position in the problem that we are trying to solve is a specific row/column combination.

One thing that is **absolutely wrong**, but I see many students in Java do is:

```
public static boolean badSolve(board, row, col) {
        /*
        * maybe some code here, maybe not
        */

        for(int row = 0; row < 9; row++) {
                for(int col = 0; col < 9; col++) {
                        if(board[row][col] == 0){
                                // find a valid number for board[row][col]
                                // do the recursive stuff
                        }
                }
        }

        /*
```

```
        * maybe some code here, maybe not
        */
        return false; // backtrack
}
```

Students employing this strategy are trying to find the next blank spot fill it with the first number that works, then recursively call solve to find the next spot. The key issue here is that they are trying to find a blank spot. The algorithm should always know where it is in the problem, which is why `row` and `col` are given as arguments. Not doing so leads to issues of erasing original parts of the puzzle between what was originally given and what you placed.

A correct algorithm has two "base" cases and two recursive cases. For our base case, we would check if the `col` is out of bounds, meaning we are done with the current row. If so, go to the next `row`. If the `row` is out of bounds, we finished all rows and can return `true` to indicate a found solution.

Our recursive cases are fairly straightforward and depend on whether `board[row][col]` is empty or not. If it has a number already in it, that is a number that was given to us as part of the puzzle, so we recursively `return solve(board, row, col+1)`. If the square is empty, we look for a valid number to put in the square, then recursively call `solve(board, row, col+1)`.

Chapter 10

Trees

Our next major data structure is trees. Specifically, we will be looking at binary search trees.

Trees are an excellent data structure for storing things since they implement all the operations we care about for collections in logarhythmic time[1]

However, trees are not without limitations. Trees will only work with data that can be stored hierarchically or in an order.

10.1 The Parts of a Tree

The first thing we need to do when introducing trees is define a vocabulary.

Much like the linked list, a tree is made of nodes. However, unlike a linked list , nodes in a tree are not arranged in a line, Instead, they are arranged in a heirachy.

Each node sit above multiple other nodes , with the nodes below it being referred to as their children or child nodes. The node connecting all these children is called the parent.

<A picture of one node, Represented by a circle with four arrows coming out below it. Each arrow points to yet another node. The Node with the arrows coming out of it is the parent, and the nodes below it are the children >

This relationship can be extended Ad infinitum as we can see with the picture below

<Picture with nodes labeled>

However anything above grandchild and grandparent just becomes tedious , so we tend to Generalize this relationship to ancestors and descendants. A key point here is to remember that while we are borrowing terms from the family tree , nodes will only have one parent . Each node can have multiple children, however .

We refer to the links connect each of the nodes as branches or links or edges. This tends to be a matter of personal preference.

[1]Specifically , Trees implement everything in average case log rhythmic time and worst case linear time , but if we do a bit of extra work and make it a self balancing binary tree (which will seem much later in this chapter) we can make this tree worst case log arhythmic for all operations

Finally , we have one special node that sits above all the other nodes . This note is the root and it is analogous to the head of a linked list . All of our operations will start at the root of the node[2].

Remember , programmers are stereotypically outdoors of averse, So they May have forgotten what a real tree looks like. Thus, we'll see that the root of the tree is at the top of the tree and our leaves are at the bottom[4]

10.1.1 Where the Recursion comes in

There is a reason we learned recursion before we introduce trees. Trees are the exemplar recursive data structure

Each tree has a root and That route has children . If we view each of those children as the root of their own subtree , this can make our algorithms for adding removing and searching extremely easy to write.

<picture Of tree, the recursive subtrees are dash circled.>

<Picture of the left subtree, with it's trees circled>

10.2 Binary Search Trees

A diagram of a binary search tree. It is made up of nodes, represented by circles, and edges (also called links or branches), represented by arrows.

10.3 Building a Binary Search Tree

10.3.1 The Code Outline

As explained in Section 9.4.3, when we use the `Comparable` class in Java to require that all objects stored in the tree has a **total ordering**, meaning every pair of objects we're storing has an ordering. In practice, this means that anything `Comparable` can be sorted.

Python, of course, doesn't need these restrictions.

```
public class BinaryTree<E extends Comparable<E>> {
}
```

Much like our Linked List, we don't need much in the way of instance variables. We'll create a `root` to keep track of the starting place for our tree and size to keep track of how many items we have stored.

Finally, we will also create our inner `Node` class for the Tree. It needs to hold the item and the locations of the left and right children. We'll also go ahead and add a The constructor and a method for printing out the item in the node (`toString` in Java and `__str__` in Python)

[2]Remember , programmers are stereotypically outdoors of averse, So they May have forgotten what a real tree looks like. Thus, we'll see that the root of the tree is at the top of the tree and our leaves are at the bottom[3]

[4]Or maybe it's some weird hydroponic zero-G kind of thing.

Listing (Java) 10.1: The Constructor and Inner Class

```
public class BinaryTree<E extends Comparable<E>> {
    private Node<E> root;
    private int size;

    public BinaryTree() {
        this.root = null;

    }

    private static class Node<E extends Comparable<E>> {
        private E item;
        private Node<E> left;  // left child
        private Node<E> right; // right child
        public Node(E item) {
            this.item = item;
        }
        public String toString() {
            return item.toString();
        }
    }


}
```

10.3.2 Contains

10.3.3 Add

All of our operations in out `BinaryTree` will be implemented recursively.

10.3.4 Delete

Chapter 11

Heaps

11.1 Priority Queues

11.2 Removing From other locations

Chapter 12

Sorting

Now that we have a handle on sorting =,

12.1 Quadratic-Time Algorithms

12.1.1 Bubble Sort

12.1.2 Selection Sort

Unlike Bubble Sort, Selection Sort has an actual use case. While the number of comparisons is always $O(n^2)$, the number of exchanges is $O(n)$. That means that we are doing only a single swap for every item we have to sort.

In other words, sorting on a computer assumes that comparisons are more expensive operation, but if that actual exchange of items is what is expensive, you should definitely consider Selection Sort. This could be the case if we are moving

12.1.3 Insertion Sort

12.2 Log-Linear Sorting Algorithms

The most commonly used sorting algorithms take $O(n \lg(n))$ time. This is the hard limit on runtime

12.2.1 Tree Sort

The tree sort is the simplest algorithm to we will cover. Performing Tree sort is a matter of three simple steps

1. Create a tree.
2. Load the items you want to sort into the tree.
3. Perform an inorder traversal of the tree.

The performance of this algorithm depends completely on the type of tree we create for this algorithm. Using a self-balancing binary search tree, adding

n items to the tree takes $O(n \lg(n))$ and an in order traversal takes $O(n)$ steps, for a grand total of $O(n)$ runtime. Using a binary search tree that does not self balance means that there is a worst case scenario of $O(n^2)$ for adding all the n items.

Using a tree also means we use extra space since all the data has to be moved into a tree, using $O(n)$ space.

12.2.2 Heap Sort

You might expect that heapsort deserves the same treatment as treesort. After all, a heap has the same structure as a tree and both are constructed to perform operations in *logn* time.

12.2.3 Heapify

12.2.4 Quick Sort

12.2.5 Merge Sort

12.3 Unique Sorting Algorithms

12.3.1 Shell Sort

The time complexity of Shell Sort is still an open problem.

12.3.2 Radix Sort

all of our prior algorithms relied on sorting items by comparing them with each other; Radix sort is unique in that no comparisons occur.

12.4 State of the Art Sorting Algorithms

12.4.1 Tim Sort

12.4.2 Quick Sort

12.5 But What if We Add More Computers: Parallelization and Distributed Algorithms

Parallel sorting algorithms are designed to be executed on a single computer with multiple processors or cores, while distributed sorting algorithms are designed to be executed on a network of computers working together. Both types of algorithms can be used to significantly improve the performance of sorting for large data sets, especially when the data does not fit in the memory of a single computer.

There are many different parallel and distributed sorting algorithms, each with its own characteristics and trade-offs. Some common techniques used in these algorithms include:

Data partitioning: Splitting the data into smaller chunks that can be sorted independently and then merged back together. Load balancing: Ensuring that

the work is distributed evenly among the available processors or computers. Communication: Allowing the processors or computers to communicate and exchange data during the sorting process.

Some examples of parallel and distributed sorting algorithms include:

Parallel merge sort: A parallel version of the merge sort algorithm that divides the data into smaller chunks and sorts them in parallel, then merges the sorted chunks back together. MapReduce: A programming model for distributed computing that is often used for sorting large data sets in a distributed environment, such as on a cluster of computers. Bitonic sort: A parallel sorting algorithm that uses a recursive divide-and-conquer approach to sort the data using a network of processors.

There are many other parallel and distributed sorting algorithms as well, each with their own specific characteristics and trade-offs. If you are interested in learning more about these algorithms, you may want to consider reading more about parallel and distributed computing, as well as specific techniques such as data partitioning, load balancing, and communication.

Parallel VS Distributed

12.6 Further Reading

12.6.1 Pedagogical Sorting Algorithms

Bogo Sort

Sleep Sort

Stooge Sort

This is primarily used as a means of testing students on using the **Master Theorem** for calculating the time complexity for algorithms.

Part IV

Hashing

Chapter 13

Sets

Sets programmed implementations of mathematical sets

13.1 Operations

We will use Venn diagrams to graphically demonstrate operates with two sets

13.1.1 Adding an item to a Set

Adding items to a set is fairly straightforward.

As we will see, adding to a set can be either $O(1)$ or $O(\log n)$ time, depending on the implementation

13.1.2 Removing an item to a Set

13.1.3 Union

In Java, this is the `addAll()` method.

13.1.4 Intersection

13.1.5 Set Difference

13.1.6 Subset

13.2 Operation Analysis

Most sets are implemented using a Hash Table.

13.2.1 TreeSet Vs HashSet Vs Linked Hash Set

13.3 Sets and Problem Solving

Sets are super efficient checklists.

13.3.1 Checking for Uniqueness or Finding Duplicates

Chapter 14

Maps

Chapter 15

Hash Tables

Our goal here is to do what seems impossible: achieve $O(1)$ lookup, insertion, and deletion of items in a collection. Fortunately, it is possible, albeit with some sacrifices:

- In order to achieve the "ultimate" time efficiency, we need to sacrifice any semblance of memory efficiency. We're still dealing with $O(n)$ space complexity, but realistically expect 33% to 50% of the space to be spent in sacrifice of our goal.
- The default way of building a HashMap will mean that we have no control over how items are stored in, meaning if we require items to be sorted or we need to keep track of any semblance of order for the inserted data, look to another structure.

15.1 Creating a Hash Function

Chapter 16

Map Reduce

16.1 Map

The `map()` operation[1] is a powerful function that may require us to think differently about the way we have approached programming so far.

The map operation takes in 2 arguments, a collection and a function to apply to every item in the collection

When we are writing functions , we are creating new verbs for our programming language to use . These verbs take in arguments, nouns that we may have declared or defined ourselves. But one thing that we May not have done yet is passing a function as an argument to another function.

This is not an uncommon operation in mathematics Example listed below

The semantics of this in every programming language is different , but the concept is the same

Why introduces here? Because a lot of common operations that can be done with map reduce involve using hash tables

[1]It is mildly confusing that there is a `map` data structure and a `map()` operation, so I will be marking the `map()` operation with a function invocation.

Part V

Relationships

Chapter 17

Graphs

In some ways, Graphs are the most important data structure. Graphs represent and model relationships, and humans are defined by relationships.

Graphs have two components: vertices (also called nodes) and edges. Graphs model the relationships between different vertices by connecting them with edges:

AN EXAMPLE GRAPH

The archtypical examples of graphs used to be maps and the distances between landmarks or looking for the shortest path.

With the advent of social media, we can talk about graphs with a few examples that might be easier to intuit.

17.1 Introduction and History

17.2 Qualities of a Graph

The physical layout of a graph doesn't actually matter[1]

17.2.1 Vertices

- Vertices must be unique.

[1]Some properies, such as whether a graph is *planar* or *bipartite* effectively care if a graph can be physically laid out in a certain way.

17.2.2 Edges

Undirected Edges

Directed Edges

Weighted Edges

17.3 Special Graphs and Graph Properties

17.3.1 Planar Graphs

Graphs that are planar can have their vertices and edges laid out in such a way that no two edges will cross.

17.3.2 Bipartite Graphs

17.3.3 Directed Acyclic Graphs

17.4 Building a Graph

17.4.1 Adjacency List

17.4.2 Adjacency Matrix

Matrix multiplication and GPU Abuse

17.5 Graph Libraries

Your programming language of choice may not

17.5.1 Java - JUNG

17.5.2 Python - networkx

There is only one realistic choice for using graphs in Python. The package networkx is extremely powerful, extremely versatile,and actively maintained.

17.6 Graphs, Humans, and Networks

17.6.1 The Small World

The Milgram Experiment

The Less-Known Milgram Experiment

17.6.2 Scale Free Graphs

17.7 Graphs in Art and Nature - Voronoi Tessellation

Figure 17.1: The wings of a dragonfly. Credit: Joi Ito (CC BY 2.0)

Chapter 18

Graph Algorithms

End of book.

Bibliography

[1] Leonardo Bonacci. Liber abaci. 1202.

[2] Python Devs. listobject.c.

[3] Phineas Mordell. *The origin of letters and numerals: according to the Sefer Yetzirah*. P. Mordell, 1914.

[4] Laurence Sigler. *Fibonacci's Liber Abaci: A translation into modern English of Leonardo Pisano's book of calculation*. Springer.

[5] Parmanand Singh. The so-called fibonacci numbers in ancient and medieval india. *Historia Mathematica*, 12(3):229–244, 1985.

[6] Susie Turner. Flowers and the fibonacci sequence. `https://www.montananaturalist.org/blog-post/flowers-the-fibonacci-sequence/`, April 2020. Accessed: 2025-06-11.

[7] Unknown - Possibly Abraham. Sefer Yetzirah.