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

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

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.

Part II

Lists

Part III

Recursion

Chapter 3

Recursion

3.1 Introduction

3.1.1 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.

3.2 Recursive Mathematics

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

3.2.1 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)[6] [2], the oldest book of Jewish Mysticism.

שבע כפּולות כיצד צרפּן. 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 אלפּים וארבעים בתים. מכאן ואילך צא וחשוב מה שאין הפּה 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) 3.1: Factorial - Iterative

```java
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 3.1. 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.

$$
\begin{aligned}
1! &= 1 \cdot 0! = 1 \\
2! &= 2 \cdot 1! = 2 \\
3! &= 3 \cdot 2! = 6 \\
4! &= 4 \cdot 3! = 24 \\
5! &= 5 \cdot 4! = 120 \\
6! &= 6 \cdot 5! = 720 \\
7! &= 7 \cdot 6! = 5040
\end{aligned}
$$

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

$$
\begin{aligned}
n! &= 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n \\
&= n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1 \qquad (3.1) \\
&= n \cdot (n-1)!
\end{aligned}
$$

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) 3.2: Factorial - Recursive

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

Listing (Python) 3.3: 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.

<Insert picture of recursive factorial calls here>

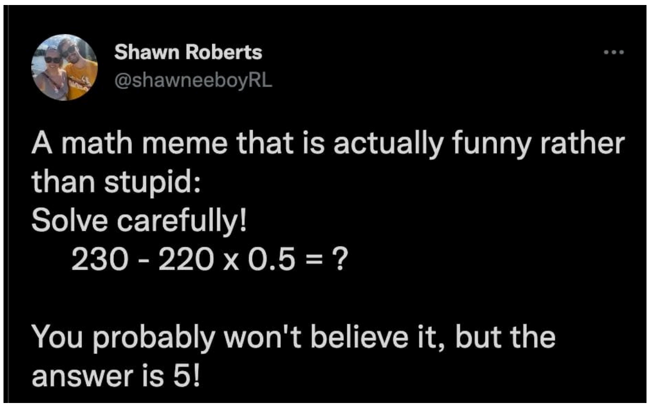
$$\begin{tikzcd}
\begin{array}{c}\texttt{factorial(4)}\\ \texttt{returns } 4\times 6 = \mathbf{24}\end{array} \arrow[d] \\
\begin{array}{c}\texttt{factorial(3)}\\ \texttt{returns } 3\times 2 = 6\end{array} \arrow[d] \\
\begin{array}{c}\texttt{factorial(2)}\\ \texttt{returns } 2\times 1 = 2\end{array} \arrow[d] \\
\begin{array}{c}\texttt{factorial(1)}\\ \texttt{returns } 1\times 1 = 1\end{array} \arrow[d] \\
\begin{array}{c}\texttt{factorial(0)}\\ \texttt{base case, returns } 1\end{array}
\end{tikzcd}$$

Figure 3.1: 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!

$$\begin{tikzcd}
\begin{array}{l}\texttt{factorial(4)}\\ \texttt{returns } 4\times 6 = \mathbf{24}\end{array} \arrow[d, "\texttt{calls factorial(3)}"] \\
\begin{array}{l}\texttt{factorial(3)}\\ \texttt{returns } 3\times 2 = 6\end{array} \arrow[d, "\texttt{calls factorial(2)}"] \\
\begin{array}{l}\texttt{factorial(2)}\\ \texttt{returns } 2\times 1 = 2\end{array} \arrow[d, "\texttt{calls factorial(1)}"] \\
\begin{array}{l}\texttt{factorial(1)}\\ \texttt{returns } 1\times 1 = 1\end{array} \arrow[d, "\texttt{calls factorial(0)}"] \\
\begin{array}{l}\texttt{factorial(0)}\\ \textbf{\texttt{base case, returns }} 1\end{array}
\end{tikzcd}$$

Figure 3.2: The call stack for **`factorial(4)`**, styled to show the flow of calls and returns.



Shawn Roberts
@shawneeboyRL

A math meme that is actually funny rather than stupid:
Solve carefully!
230 - 220 x 0.5 = ?

You probably won't believe it, but the answer is 5!

Figure 3.3: Hopefully you get it now.

3.2.2 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) 3.4: Recursion with no end condition - Java

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

Listing (Python) 3.5: Recursion with no end condition - Python

```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.

3.2.3 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.

[2]maybe

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, 3]. 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 [4] 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 [5].

Implementation

Implementing this as a recursive function is rather trivial!

Listing (Java) 3.6: Naive Java Implementation

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

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

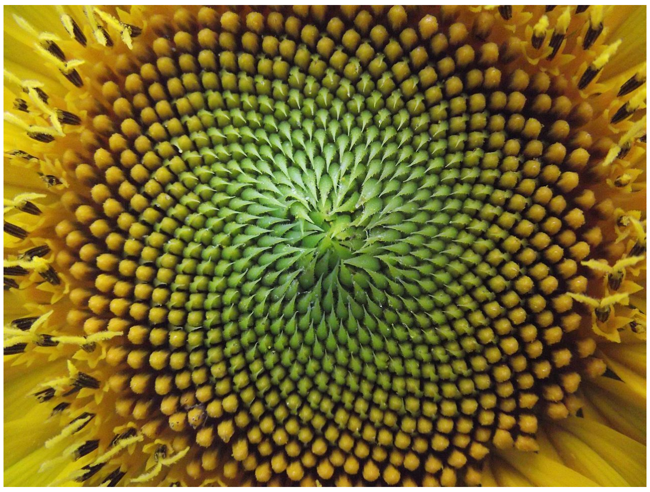


Figure 3.4: The sunflower's fibonacci spiral. Photo by Anna Benczur, CC by-SA 4.0.

Listing (Python) 3.7: 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.

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)`

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."

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

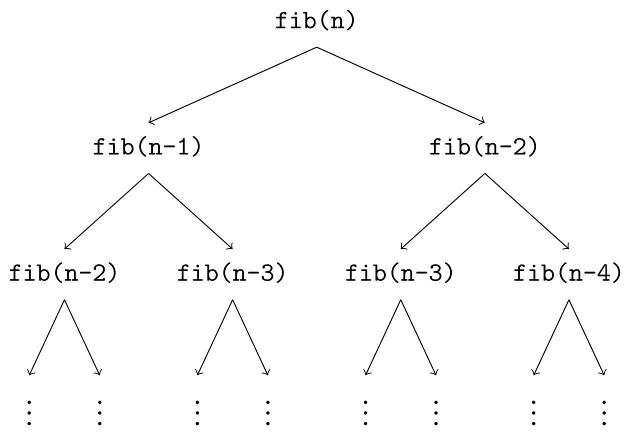


Figure 3.5: Recursive Function Calls for `fib(n)`. Notice that the call to `fib(n-1)` must independently compute `fib(n-2)`, thus duplicating a ton of work.

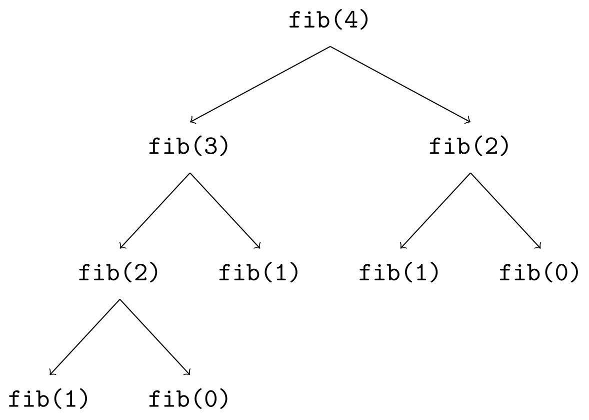


Figure 3.6: Computing `fib(4)`.

Listing (Java) 3.8: An Efficient Recursive Fibonacci Algorithm

```java
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 cal-

Listing (Python) 3.9: An Efficient Recursive Fibonacci Algorithm in Python

```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)`.

3.3 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.

3.3.1 Printing Recursively

Listing (Java) 3.10: Recursive Printing: Java

```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) 3.11: Recursive Printing: Python

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

3.4 Arrays with Recursion

3.4.1 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.

3.4.2 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) 3.12: 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) 3.13: Recursive Linear Search - Python

```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.

3.4.3 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 3.7.

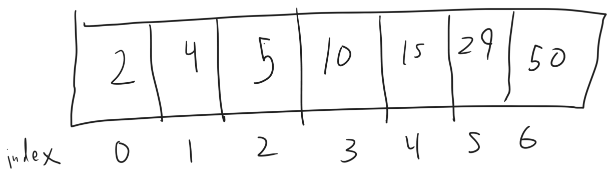


Figure 3.7:

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 3.8).

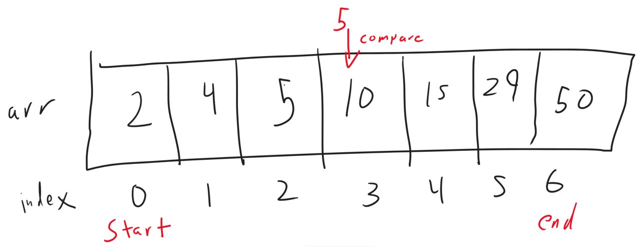


Figure 3.8: The labels `start` and `end` represent the start and end of our search space. This will make more sense as we progress, especially as we start coding.

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 3.9). 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 3.10).

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.

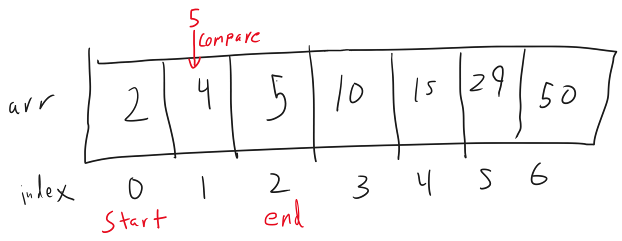


Figure 3.9: The labels **start** and **end** represent the start and end of our search space, which has now shrunk to less than half the original array.

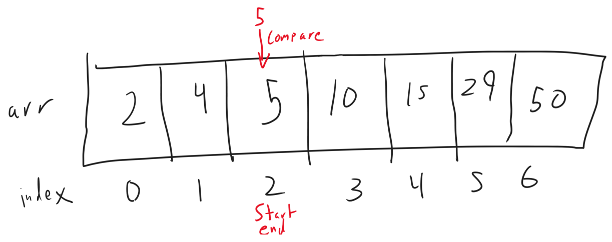


Figure 3.10: The labels **start** and **end** are now on the same item. This means a search space has a size of 1.

Code

Listing (Java) 3.14: Binary Search - Java

```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) 3.15: Binary Search in Python

```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 nears 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 more 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:

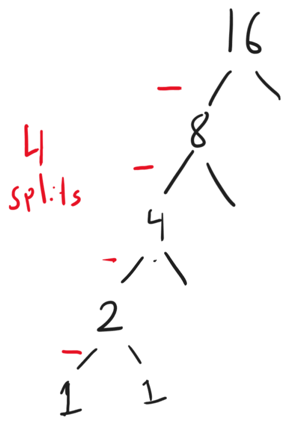


Figure 3.11:

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) 3.16: Binary Search - Java Lists

```java
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

3.5 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.

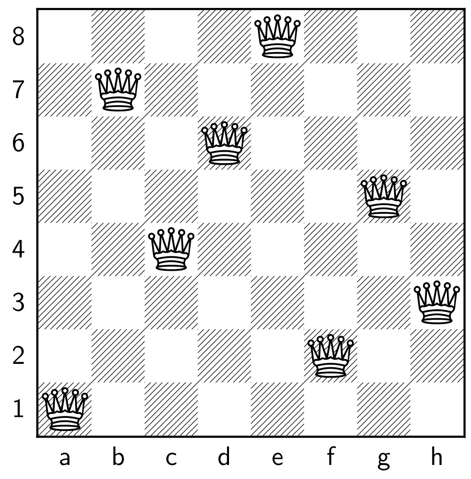
Teachers: Give these exercises, or something similar like the knight's tour, as a homework assignment. These are engaging and worth it. Yes, some students are going to completely ignore the plea of the last paragraph, but they were going to do that no matter what assignment you gave. Don't rob the engaged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

3.5.1 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.

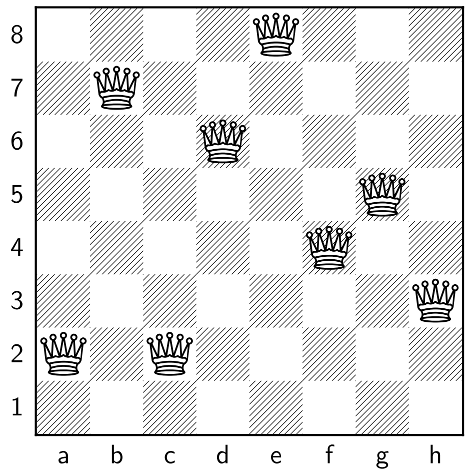
The goal of the puzzle is to place eight queens on a standard 8×8 chessboard. The queens should be placed in such a way that no queen can capture any other queen. In chess, a queen can capture any piece by traveling vertically, horizontally, or diagonally until it crashes into a piece. Or in more programming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

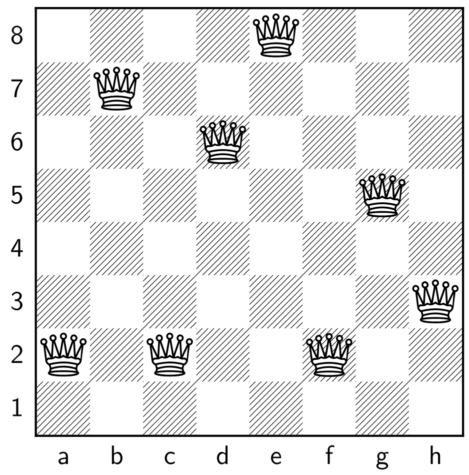
This is one example solution, but it is not the only solution.



Brute Force Solution

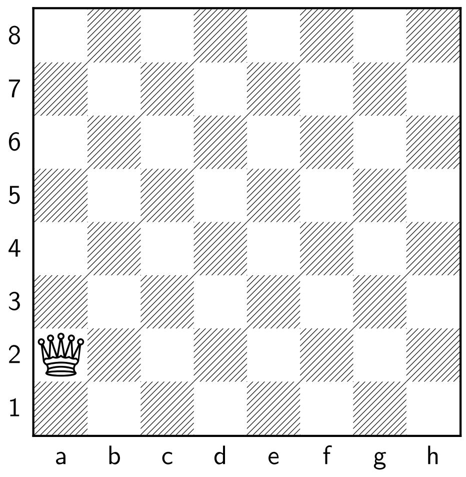
A brute force algorithm means we will be checking every single possible state to find a solution. In this case, a brute force solution for the Eight Queens Puzzle would every possible placement of eight queens on a chessboard, such as these two incorrect solutions:

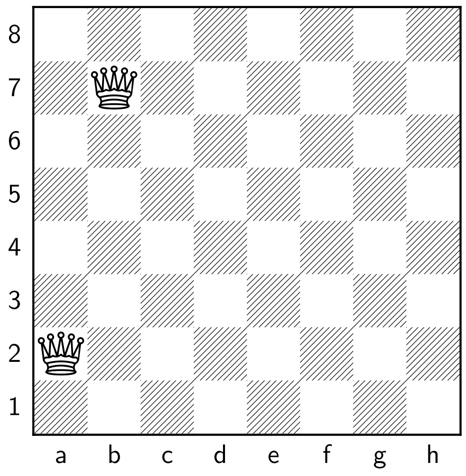


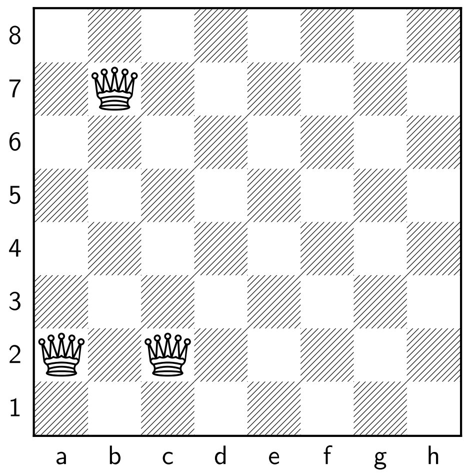


There are a total of $\binom{64}{8} = 4426165368$ possible ways to place 8 queens on a chessboard with 64 spaces. Our computer could sift thru all possible configurations until it finds a solution, and thus, performing a brute force solution will eventually work. It just won't work fast.

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 **??**, 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.

3.5.2 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) 3.17: 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) 3.18: 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) 3.19: Solve - 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(board, row, col)){
            board[row][col] = "Q";
            if(solve(board, pos + 1) == true){
                return true;
            }
            board[row][col] = "-";
        }
    }
    return false; // backtrack
}
```

Listing (Python) 3.20: Solve - Python

```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.

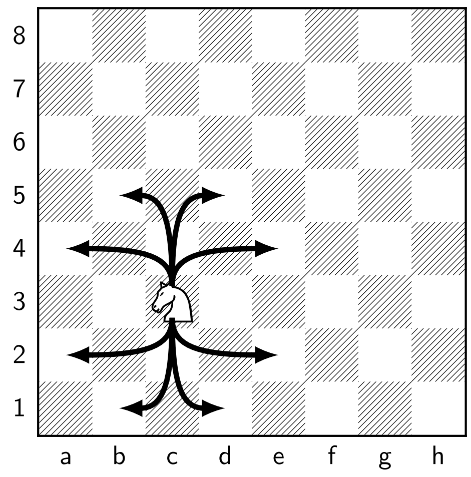
My advice is to work on the first case and test that. If it works, you'll get a line of Queens going from the top left corner to the bottom right.

3.5.3 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)`

Part IV

Hashing

Part V

Relationships

Bibliography

[1] Leonardo Bonacci. Liber abaci. 1202.

[2] Phineas Mordell. *The origin of letters and numerals: according to the Sefer Yetzirah.* P. Mordell, 1914.

[3] Laurence Sigler. *Fibonacci's Liber Abaci: A translation into modern English of Leonardo Pisano's book of calculation.* Springer.

[4] Parmanand Singh. The so-called fibonacci numbers in ancient and medieval india. *Historia Mathematica*, 12(3):229–244, 1985.

[5] Susie Turner. Flowers and the fibonacci sequence. `https://www.montananaturalist.org/blog-post/flowers-the-fibonacci-sequence/`, April 2020. Accessed: 2025-06-11.

[6] Unknown - Possibly Abraham. Sefer Yetzirah.