Data Structures

Andrew Rosen

Contents

I Preliminaries **7**

1 Introduction **9**
1.1 What is a Data Structures Course 9
1.2 Why This Book? 9
1.2.1 Where Does This Book Fit Into a Computer Science Curriculum 9
1.2.2 What Are My Base Assumptions about the Reader? . . . 10
1.3 To The Instructor 10
1.3.1 Professor Rosen's Extremely Opinionated Advice on How to Lecture 10
1.3.2 Exercises 11
1.3.3 The Order 11
1.3.4 Assignments 11
1.3.5 How to Use 11
1.4 To The Student 11
1.4.1 How to use 12
1.5 License 12
1.6 On Styles 12

II Lists **13**

III Recursion **15**

2 Recursion **17**
2.1 Introduction 17
2.1.1 Why? 17
2.2 Recursive Mathematics 17
2.2.1 Factorial 17
2.2.2 Recursive Rules 21
2.2.3 Fibonacci 22
2.3 More Examples 26
2.3.1 Printing Recursively 26
2.4 Arrays with Recursion 27
2.4.1 Summation of an Array 27
2.4.2 Recursive Linear Search 27
2.4.3 Binary Search 28

2.5 Recursive Backtracking . . . 33
2.5.1 Mazes Again . . . 34
2.5.2 The Eight Queens Puzzle . . . 34
2.5.3 Additional Problems left to the Reader . . . 35
2.6 Recursive Combinations . . . 35
2.7 Recursion and Puzzles . . . 35
2.8 Recursion and Art . . . 35
2.9 Recursion and Nature . . . 35

3 Trees **37**
3.1 The Parts of a Tree . . . 37
3.1.1 Where the Recursion comes in . . . 38
3.2 Binary Search Trees . . . 38
3.3 Building a Binary Search Tree . . . 38
3.3.1 The Code Outline . . . 38
3.3.2 Add . . . 39
3.3.3 Contains . . . 39
3.3.4 Delete . . . 39

4 Heaps **41**
4.1 Priority Queues . . . 41
4.2 Removing From other locations . . . 41

5 Sorting **43**
5.1 Quadratic-Time Algorithms . . . 43
5.1.1 Bubble Sort . . . 43
5.1.2 Selection Sort . . . 43
5.1.3 Insertion Sort . . . 43
5.2 Log-Linear Sorting Algorithms . . . 43
5.2.1 Tree Sort . . . 43
5.2.2 Heap Sort . . . 44
5.2.3 Heapify . . . 44
5.2.4 Quick Sort . . . 44
5.2.5 Merge Sort . . . 44
5.3 Unique Sorting Algorithms . . . 44
5.3.1 Shell Sort . . . 44
5.3.2 Radix Sort . . . 44
5.4 State of the Art Sorting Algorithms . . . 44
5.4.1 Tim Sort . . . 44
5.4.2 Quick Sort . . . 44
5.5 But What if We Add More Computers: Parallelization and Distributed Algorithms . . . 44
5.6 Further Reading . . . 45
5.6.1 Pedagogical Sorting Algorithms . . . 45

IV Hashing **47**

6 Sets **49**
6.1 Operations . . . 49

6.1.1 Adding an item to a Set . . . 49
6.1.2 Removing an item to a Set . . . 49
6.1.3 Union . . . 49
6.1.4 Intersection . . . 49
6.1.5 Set Difference . . . 49
6.1.6 Subset . . . 49
6.2 Operation Analysis . . . 49
6.2.1 TreeSet Vs HashSet Vs Linked Hash Set . . . 49
6.3 Sets and Problem Solving . . . 49
6.3.1 Checking for Uniqueness or Finding Duplicates . . . 50

7 Maps **51**
7.1 What is a Map . . . 51
7.2 Functions . . . 51
7.3 Costs . . . 51
7.3.1 Tree-Based Map . . . 51
7.3.2 Hash Table Map . . . 51
7.4 Streams, List Comprehensions, and Collectors . . . 51

8 Hash Tables **53**
8.1 Creating a Hash Function . . . 53

9 Map Reduce **55**
9.1 Map . . . 55

V Relationships **57**

10 Graphs **59**
10.1 Introduction and History . . . 59
10.2 Qualities of a Graph . . . 59
10.2.1 Vertices . . . 59
10.2.2 Edges . . . 60
10.3 Special Graphs and Graph Properties . . . 60
10.3.1 Planar Graphs . . . 60
10.3.2 Bipartite Graphs . . . 60
10.3.3 Directed Acyclic Graphs . . . 60
10.4 Building a Graph . . . 60
10.4.1 Adjacency List . . . 60
10.4.2 Adjacency Matrix . . . 60
10.5 Graph Libraries . . . 60
10.5.1 Java - JUNG . . . 60
10.5.2 Python - networkx . . . 60
10.6 Graphs, Humans, and Networks . . . 60
10.6.1 The Small World . . . 60
10.6.2 Scale Free Graphs . . . 60
10.7 Graphs in Art and Nature - Voronoi Tessellation . . . 60

11 Graph Algorithms **63**
11.1 Searching and Traversing 63
11.1.1 Breadth First Search 63
11.1.2 Depth First Search 63
11.2 Shortest Path 63
11.2.1 Djikstra's Algorthim 63
11.2.2 Bellman-Ford 63
11.3 Topological Sorting 63
11.3.1 Khan's Algorithm 63
11.4 Minimum Spanning Trees 63
11.4.1 Kruskal's Algorithm 63
11.4.2 Prim's Algorithm 63

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.

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.

Part II

Lists

Part III

Recursion

Chapter 2

Recursion

2.1 Introduction

2.1.1 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.

2.2 Recursive Mathematics

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

2.2.1 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 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) 2.1: 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 2.1. 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.

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n \\ &= n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1 \\ &= n \cdot (n-1)! \end{aligned} \tag{2.1}$$

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) 2.2: Factorial - Recursive

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

Listing (Python) 2.3: 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>

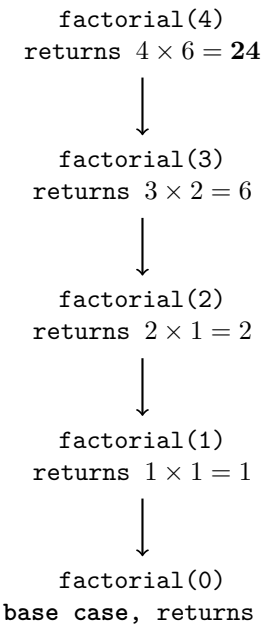


Figure 2.1: The call stack for `factorial(4)`. Each call must wait for the result of the call below it. Once `factorial(0)` returns 1, the results are multiplied back up the stack.

[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!

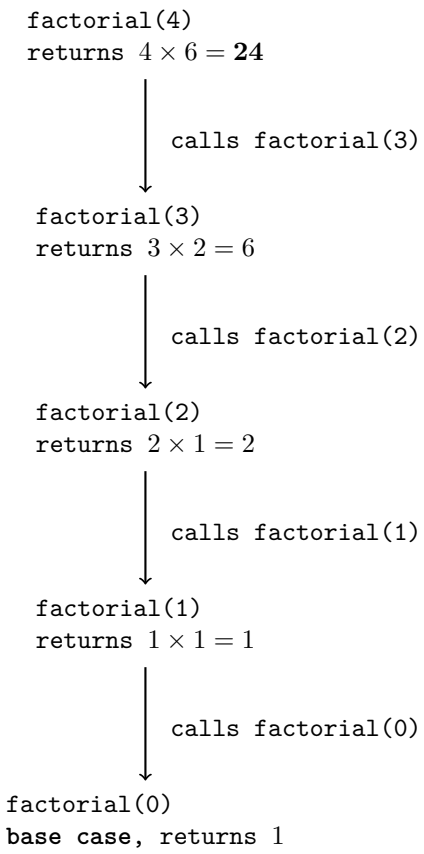


Figure 2.2: The call stack for `factorial(4)`, styled to show the flow of calls and returns.

2.2.2 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

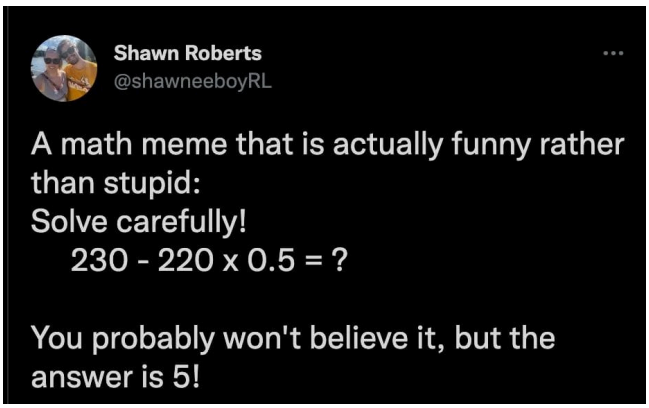


Figure 2.3: Hopefully you get it now.

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) 2.4: Recursion with no end condition - Java

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

Listing (Python) 2.5: 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.

2.2.3 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.

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) 2.6: 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) 2.7: Naive Python implementation

```python
def fib(n):
    if n == 0 or 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.

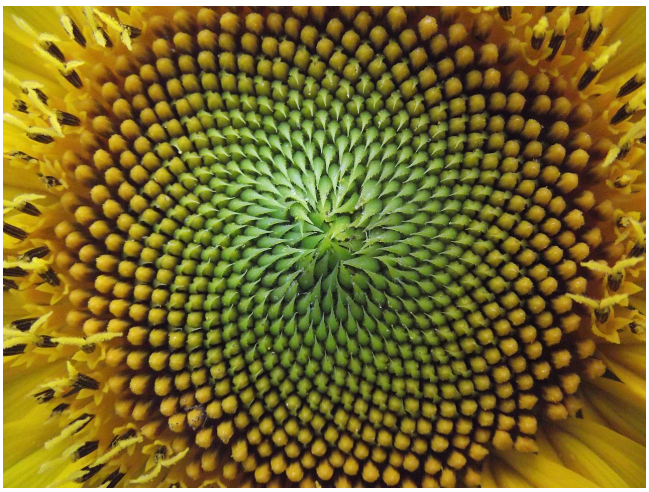


Figure 2.4: The sunflower's fibonacci spiral. Photo by Anna Benczur, CC by-SA 4.0.

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.

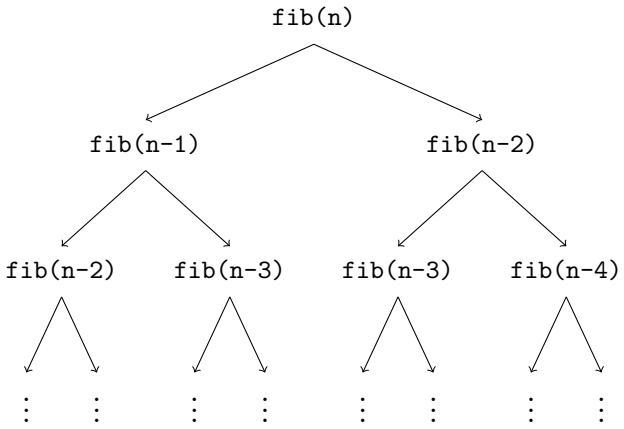


Figure 2.5: Recursive Function Calls for `fib(n)`. Notice that the call to `fib(n-1)` must independently compute `fib(n-2)`, thus duplicating a ton of work.

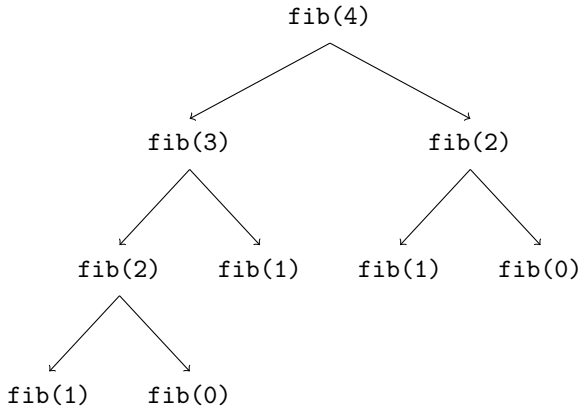


Figure 2.6: Computing `fib(4)`.

Listing (Java) 2.8: 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 nth Fibonacci number and a private helper method to do the

Listing (Python) 2.9: 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)`.

2.3 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.

2.3.1 Printing Recursively

Listing (Java) 2.10: 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) 2.11: Recursive Printing: Python

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

2.4 Arrays with Recursion

2.4.1 Summation of an Array

ADD REFERENCE HERE

We will begin our investigation with a problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of an array of numbers such as:$[1, 3, 5, 7, 9]$ An iterative method that computes the sum is shown in Listing 4.3.1. The method uses an accumulator variable (theSum) to compute a running total of all the numbers in the array by starting with and adding each number in the array.

2.4.2 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) 2.12: 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) 2.13: 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 $O(n)$ the `get` method incurs. Use the built-in iterator instead, i.e. use a for each loop.

2.4.3 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 2.7.

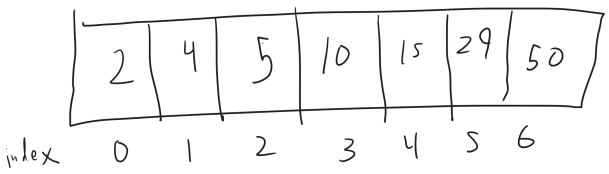


Figure 2.7:

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 2.8).

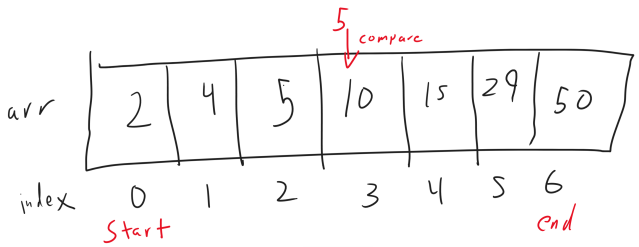


Figure 2.8: The labels `start` and `end` represent the start and end of our search space. This will make more sense as we progress, especially as we start coding.

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 2.9). 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 2.10).

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.

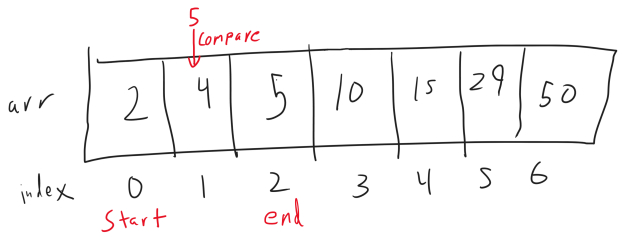


Figure 2.9: The labels `start` and `end` represent the start and end of our search space, which has now shrunk to less than half the original array.

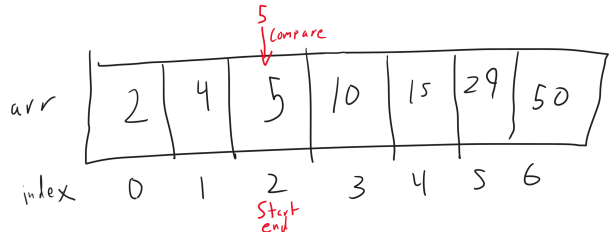


Figure 2.10: The labels `start` and `end` are now on the same item. This means a search space has a size of 1.

Code

Listing (Java) 2.14: 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) 2.15: 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. The first base case returns texttttNone to represent a failure to find.

Runtime Analysis

Each step of binary search eliminates either exactly or almost exactly half of the search space or sucessfully terminates the search.

This halving at each step is represented by $log_2(n)$, where n is the number of items. 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. Compare that to our linear search, which starts at the beginning at goes thru 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 time the algorithm takes.

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:

In algorithm analysis, *logn* in the time complexity is used to indicate that the search space gets split in half. In the Binary Search algorithm above, we 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.

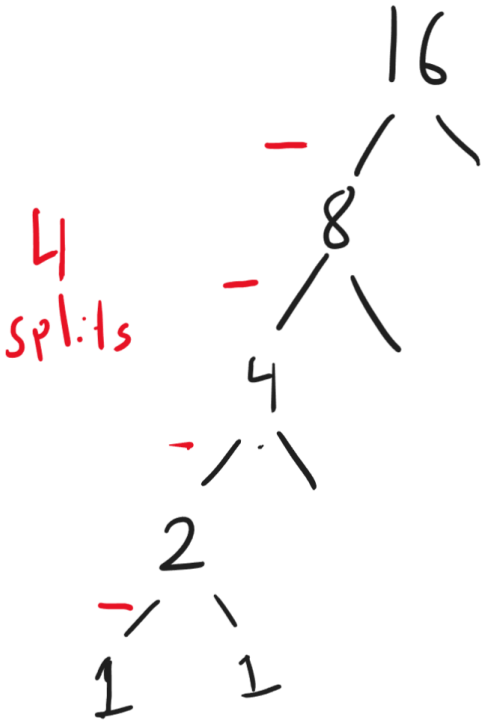


Figure 2.11:

Additional Implementation: Java with Lists

Listing (Java) 2.16: 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);
    }
}
```

2.5 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

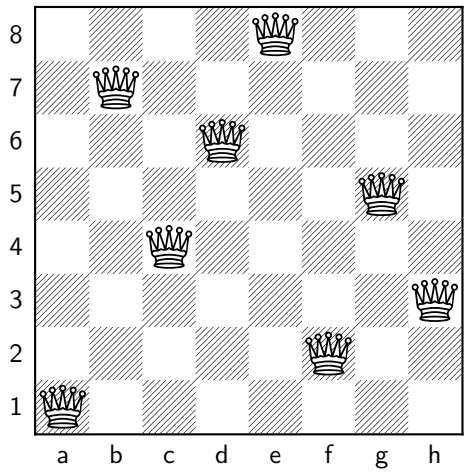
The Recursive Backtracking Algorithm

```
boolean solve(board,  pos){

        if( pos is such that there is nothing left to solve){
                return true;
        }

        for each possible choice {
                if(valid(choice)){
                        mark board at pos with choice;
                        if(solve(board, pos + 1) == true){
                                return true;
                        }
                        unmark board at pos if needed, as choice was invalid
```

```
                }
        }
        clear any choices entered at pos on board, if needed;

        return false; // backtrack
}
```

2.5.1 Mazes Again

2.5.2 The Eight Queens Puzzle



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:

<Chess notation here>

There are a total of $\binom{64}{8} = 4426165368$ possible ways to place 8 queens on a chessboard with 64 spaces.

Recursive Solution Outline

```
public static boolean solve(int[][] board, int col){

        if( col == 8){
                return true;
        }

        for(int row = 0; row < 8;row++){
                if(valid(choice)){
```

```
                        mark board at pos with choice;
                        if(solve(board, pos + 1) == true){
                                return true;
                        }
                        unmark board at pos if needed, as choice was invalid
                }
        }
        clear any choices entered at pos on board, if needed;

        return false; // backtrack
}
```

A Place Holder For Validity

Performing the Recursion

Checking just One condition

Checking all the Conditions

2.5.3 Additional Problems left to the Reader

Knight's Tour

Sudoku

2.6 Recursive Combinations

2.7 Recursion and Puzzles

2.8 Recursion and Art

2.9 Recursion and Nature

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.