Chapter 5

Algorithms and Beyond

In this section we are going to be covering three main bases of Algorithms: ***Concepts, Representations, and Structures***. After, we will explore beyond this into different types of algorithms. There is a lot to cover with algorithms and we can't cover everything, but this will lay a strong foundation for your eventual implementation of algorithms into real programming.

Algorithms

More likely than not you've heard the term a*lgorithm* thrown around in conversation, but many people don't understand exactly what it is. An **algorithm is a defined set up steps for a computer program in order to do a task**.

An algorithm for making a Grilled Cheese sandwich could go as follows:

1. Get Ingredients (Cheese, Butter, and Bread)
2. Put cheese between two bread slices
3. Put butter on outside of bread
4. Grill
5. Eat and Enjoy

That's a fairly simple, realistic example of an everyday algorithm, but computers follow steps like these to accomplish certain tasks. Computer algorithms must be **well ordered if a computer should follow them and achieve the right outcome**. This doesn't necessarily mean the steps of an algorithm must be followed each step directly after another, but rather the **steps are thoughtfully and intelligently ordered and created**.

Algorithms **must have achievable and viable steps**. For instance, to make an algorithm that lists all numbers with a multiple of three, well that list would go on forever and thus the algorithm isn't able to achieve the task.

Something very important to note is that **Algorithms are abstract**, meaning an algorithm can be written, or represented, in a plethora of ways but all follow the same idea, the idea of what the algorithm should do. This leads us into representation, how we create and represent algorithms.

Representation of Algorithms

Algorithms are often represented through primitives and pseudocode. To start, let's delve into primitive representation.

Primitive representation is really just using language or informative blocks to show what an algorithm is doing. **A primitive is a block (language like English or a symbol) that has a specific syntax, what it looks like, and semantics, what it means, to it**. To give an example, below is a primitive block.

1. **Get a list of names of the class**
2. **Organize them in alphabetical order**
3. **Return the organized list to the teacher**
4. **Stop and be prepared to take the next list**

It's important to note that an algorithm MUST have these traits: it can and will execute properly, it has an end result, and has a clear and ordered instructions to follow. Whether or not you design an algorithm using primitives, pseudocode, or actual code, these are crucial elements to an algorithm. This algorithm above meets those criteria but is relatively simple. With more complex algorithms that well look at later and ones you make yourself, be careful not to veer away from the fundamentals of a good algorithm.

As we can see, this is a rather basic list to follow. There are other ways we can make primitive algorithms, for instance origami instruction books show people where to fold with dotted lines. The dotted lines and the pictures to show readers where and when to fold is an example of using blocks of information (semantics and syntax) to describe the algorithm. Real world examples like those show the use of primitive representation, but most common for code, is pseudocode.

Pseudocode is a step below programming, it is a system to express the ideas and process of an algorithm. A key element of pseudocode is to be precise another to have the algorithm properly translated to code but written vague enough to be translated to any

programming language. Thus, the notation must be clear and concise

Let's start writing then!

Pseudocode borrows a lot from primitives, but it's more rigorous. To begin, let's work on assigning statements. If we have a grocery cart, we will need to assign a place for our food items to go. The way we write that is

GroceryCart ← item

The ← is the assignment pseudocode operator. This says we are assigning item to our cart. Furthermore, this can be extended to basically saying something equals another, like:

Paycheck ← Money Earned - taxes

This pseudocode says a paycheck equals our money we earned but minus federal taxes. Bummer.

So, let's go farther and start working with conditions. Let's say a restaurant employee is waiting tables. If she does a good job, she gets a big tip from the customer. But, if she does a bad job, then she gets no tip. Let's look at the code below:

if(waiter does good job **) then (**tip is big**)**

else there is no tip

We can further rewrite this to look more like code with:

if(waiter does a good job **)**

then (tip = big**)**

else

(tip = nothing **)**

So, what if we want our algorithm to run multiple times on a condition? For that we can use what's called a while statement. It goes like this:

while (I am broke) **do (**look for a job and be sad**)**

Or

while (Food is in stock**) do (**sell food**)**

These are examples of how to write an algorithm that repeats a task until the condition, states after the while, is met.

We can also state procedures for the algorithm to run. We don't necessarily need to write the entire procedure if that isn't what the algorithm is supposed to do. For example:

If (I just woke up**)**

then (procedure takeBath, procedure brushTeeth, procedure makeBreakfast)

… alternatively

procedure sort (list of chores **)**

The above **pseudocode shows how we don't need to tell our algorithm *how* to take a bath, but to run the procedure or algorithm that does**. This algorithm is designed to do all those tasks on a certain condition not to describe those tasks, essentially.

These are the essentials to pseudocode, and other statements and conditions can easily be conceived and relayed by using primitives in your pseudocode. So, now let's move on to Structures.

Algorithmic Structures

There are many different algorithmic structures, many of which sort and search lists and whatnot. We will be looking at a specific few: Linear Searching, Binary Searching, Insertion Sorting.

Insertion Sorting

There are many different algorithms for sorting (i.e. Merge, Selection, Bubble, etc.). **Insertion Sorting is the simplest, all this does is compare a number on a list, sees if it is bigger, and if it is it moves up the list**. Once that number has been checked until it can't climb the list any higher, the sort moves on to the next number.

Example:

Insertion Sort						
List Order	1st	2nd	3rd	4th	5th	6th
Original list	58	26	13	51	28	99
Pass 1	26	58	13	51	28	99
Pass 2	26	13	58	51	28	99
Pass 3	13	26	51	58	28	99
Pass 4	13	26	51	28	58	99
Pass 5	13	26	28	51	58	99

So, after the first pass through the original list, it compares 58 with 26. Because 58 is bigger than 26, 58 replaces its position and 26 is sent back. The second pass has 58 replace 13. The third pass tests two different places: 13 and 26, and 58 and 51. 26 replaces 13's position and 13 is sent to the back of the list while simultaneously 58 replaces 51's position. During the 4th pass, 26 compares and cannot replaces 51 so it stays in the same place, while 58 replaces 28's position. Then in the 5th, 51 takes 28's position and 58 can no longer progress. This is how the insertion algorithm works, by inserting numbers into other places in a list.

Selection Sort

This algorithm sorts list by **finding the smallest item in a list and putting it in the right place**. It repeats this process until the list is sorted from least to greatest. Example:

Selection Sort						
List Order	1st	2nd	3rd	4th	5th	6th
Original List	58	26	13	62	21	99
Pass 1	13	26	58	62	21	99
Pass 2	13	21	58	62	26	99
Pass 3	13	21	26	62	58	99
Pass 4	12	21	26	58	62	99
Pass 5	12	21	26	58	62	99
Pass 6	12	21	26	58	62	99

As we can see, this algorithm finds the smallest number, and put it at the front of the list each iteration through. The **algorithm swaps the number it finds with the number at the start of the unordered section of the list, the darker part**. After each pass, the list of unordered numbers decreases until eventually all the numbers

are ordered. Even though bypass 5 the list is ordered, this algorithm must make sure it is ordered by finishing the rest of the list, there isn't room for mistakes in the algorithm.

Merge Sort

Merge Sorting is the most complex algorithm we'll be looking at. Often called the "Divide and Conquer" method, **Merge Sorting sorts a list into two groups until it can't anymore, then merges the groups and sorts them as they merge**. Merge sorting is best used on large groups and can be faster than Selection and Insertion Sorting.

Example:

Sort the list of numbers: 9, 5, 1, 2, 8, 3, 5, 4

Pass 1: Split the group into two: (9, 5, 1, 2) and (8, 3, 5, 4)

Pass 2: Split each group further: (9 and 5), (1 and 2), (8 and 3), and (5 and 4)

Pass 3: Split until we can no longer: 9 and 5 and 1 and 2 and 8 and 3 and 5 and 4

Pass 4: Combine and sort: (5 and 9), (1 and 2), (3 and 8), (4 and 5)

Pass 5: Combine and sort: 1, 2, 5, 9 and 3, 4, 5, 8

Pass 6: Combine and Sort: 1, 2, 3, 4, 5, 8, 9

The pseudocode and actual code for this gets very complicated. The downside to this sorting algorithm is that is uses quite a bit of computer memory. Its primary creation method uses intense programming techniques with recursion and is best left for later units to learn the nitty gritty. For now, it's important you understand *how* this sorts lists.

Linear and Binary Searching

Now that we know how to sort lists, we can search them. There are different types of searching but we are going to focusing on **Linear Searching.** Linear searching is closes to how we normally search, by going through a list and checking to see if each individual object, one by one, is the one we are looking for. Example:

```
nameList = Leon, Barry, Seika, Miles, Lachan

For every name in nameList ( if( neededName equals Seika   )

    then ( say "she seems pretty cool" )  )
```

This goes through every name in the list until it reaches the name Seika and prints something out. That's the basis of this algorithm.

The **Binary Search Algorithm** is a type of a linear search, but this is the most effective method to search for an item *in an already sorted list*. The last part is important to note, you'll see why soon. This algorithm **repeatedly guesses the middle of each list and sees if**

that guess was too high or too low. This cuts the list in half with every guess until the correct number or item is found. Example:

Our Number: 82	Guess	Response
First Guess	50	Low
Second Guess	75	Low
Third Guess	87	High
Forth Guess	81	Low
Fifth Guess	84	High
Fifth Guess	83	High
Sixth Guess	82	Correct

As we can see, Binary Search cuts the list in half with each guess, limiting the number of possible guesses by a significant amount. Look at the pseudocode below:

While (number isn't found)

If (Guess < half of the current list**)**

Then eliminate numbers lower than our current guess in the current list

Else if (Guess > half of the current list **)**

Then eliminate numbers greater than our guess in the current list

Even in pseudocode, this algorithm in kind of complicated. But, it's still important to know. If the list of number was out of order, our program wouldn't work as it might get confused and cut out the top half of a list even if the number it needs is 20 and that 20 is located in the top half of the list of 100. So, how do we make sure our list is able to be properly searched?

Algorithm Efficiency

While computers today can easily process millions of tasks in seconds, it's important that we build efficient algorithms still. In this section we will be going over what makes an algorithm efficient and what doesn't. Let's start with a problem:

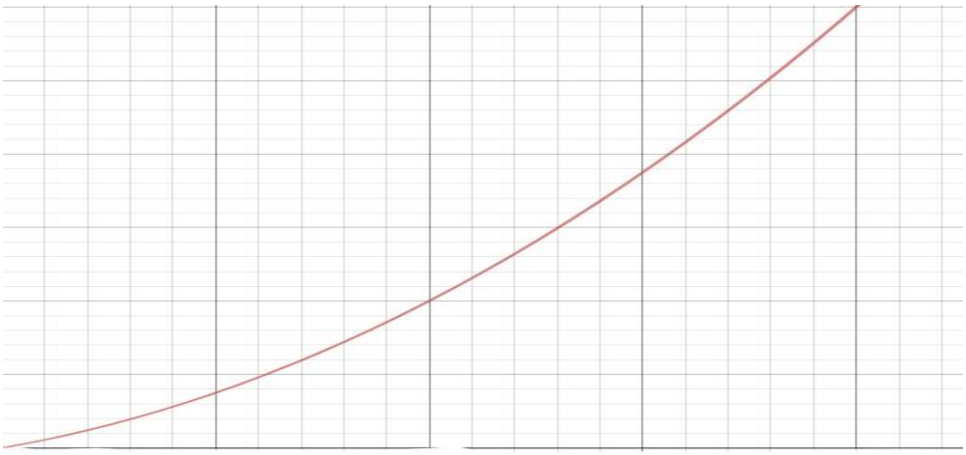
A student is doing research for a book she's writing about how evolution has caused some animals to have no necks. So, she consults good ol' Google for help! When she looks up "*Animals with no necks*", let's assume Google has one site even remotely related to that, but how might it go about searching and returning that site?

Let's say Google is using an insertion sort algorithm. It goes through each item on its roster of sites and checks if it contains the same phrase. Let's assume for a moment Google has a total of 50,000 sites to offer people. Our algorithm can compare a site for a keyword in about 10 milliseconds (or 10 one-thousandths of a second). So, on average, Google can get us the sites we need in about 500 seconds, or just a bit more than 8 minutes using the insertion sort algorithm.

Even if our student had no social life, this is still too long a wait for her. So, what else can Google do?

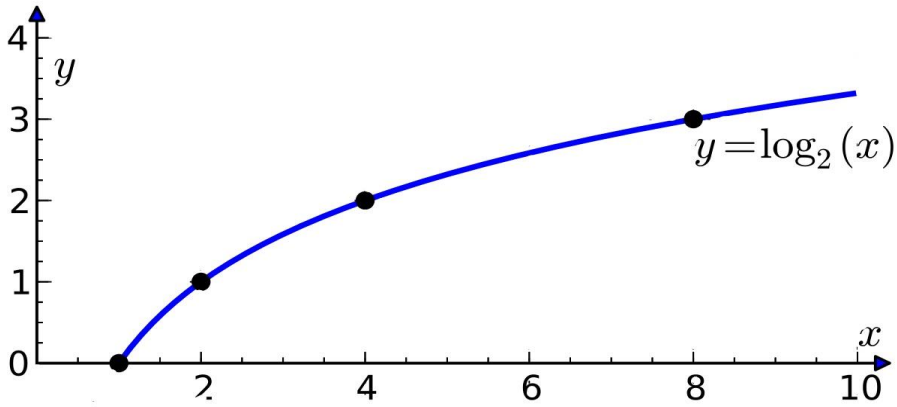
Let's try using the binary search. Again, we'll assume Google has about 50,000 sites, but now we cut the list in half. After the second inquiry of the list we only have 25,000 sites. The third gives us 12,500 sites and so on. We can see the site we're looking for is able to be found after 50 searches. With each retrieval of info from the search taking 10 milliseconds again, it takes Google about half of a second (0.5 seconds) to retrieve our information! Clearly in this situation Google should use the Binary Search Algorithm. This is what Algorithm efficiency is, and why it's so important. If we were given another scenario where the answer we were looking for was at the beginning of a list, then yes insertion sorting is going to be faster than binary searching.

In examination of algorithms each can produce a formula that verifies its efficiency as time and length progresses. By evaluating worst-case, average-case, and best-case scenarios, programmers can derive the formula for an algorithm's efficiency. Insertion sorting has an **efficiency curve of (½)(n^2 - n).**



This graph shows how the general curve insertion sorting follows. We can see how as time progresses, the less efficient the algorithm becomes.

The graph of the efficiency of a binary search algorithm is: Log n (log base 2 of n). This shows how as time goes on, the efficiency of the graph increases because the time it takes to search a list becomes more linearly horizontal.



Finally, the notation programmers use to identify and communicate different classes of algorithms is called **big-theta notation**. This is important to for communication, as graphs following a parabolic shape, like our insertion sort, are represented

as ⊖**(n^2)** or read as "big theta of n squared". The binary search can be represented at ⊖**(log n)**, read as "Big theta log of n" (Note: This is log base 2 of n). We can see at later times, ⊖**(log n)** supersedes ⊖**(n^2) as the more efficient algorithm.**

These are the basics of algorithm efficiency and are essential to algorithm correctness.

Big-O Notation

To conclude this section, let's look at the ever-classic, Big-O Notation. **Big O notation is a variation of big-theta notation to represent the complexity of a problem**. Specifically, this describes the worst-case scenario for a problem, which would take the most time or memory/disk space by an algorithm.

Let's take the example ⊖**(log n),** this runs a worst-case scenario for an algorithm. This doesn't apply to *all* the algorithm, it just shows efficiency for a certain scenario, the worst case one that it. If an algorithm gets the answer on the first guess, this is called ⊖**(1)**.