Table of Contents

Basics

Moore’s Law

Moore’s Law: the observation that over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years. [Wikipedia]

This may not mean a lot to you right now, but this “law” was an observation by the co-founder of Intel Gordon E. Moore in 1965, before personal computers were even thought of. His prediction, which he originally stated would remain true for just ten years after he wrote his paper, is to this day an insanely accurate prediction of the future of hardware, as even chips made today adhere to it.

Binary

Computers see don’t see things like people do. While they have grown advanced in the last 30 years, there are core aspects of computers that have stayed constant throughout the entire evolution of the computer. One of those things is how computers see things. In computers, everything is represented using zeros and ones, a number system known as **binary**.

Binary representation of numbers use **bits** to store information. Each one of these bits has two possible values that it can be: 0 or 1. While one bit is not by itself useful, many bits used together can represent anything your computer wants to. Everything, from the app you are currently using to a phone contact stored in your phone, is made up of a series of zeros and ones, long strings of binary numbers used to represent all the data you ever see on a computer.

How is binary useful? It may make sense to computers, but not very much so to humans. Binary is a base-2 numbering system, meaning each value of binary can have either two values. Humans use a base-10 numbering system, meaning each value of a number can have 10 values, 0 through 9.

Let’s take a look at how the number 29 is represented in base-10.

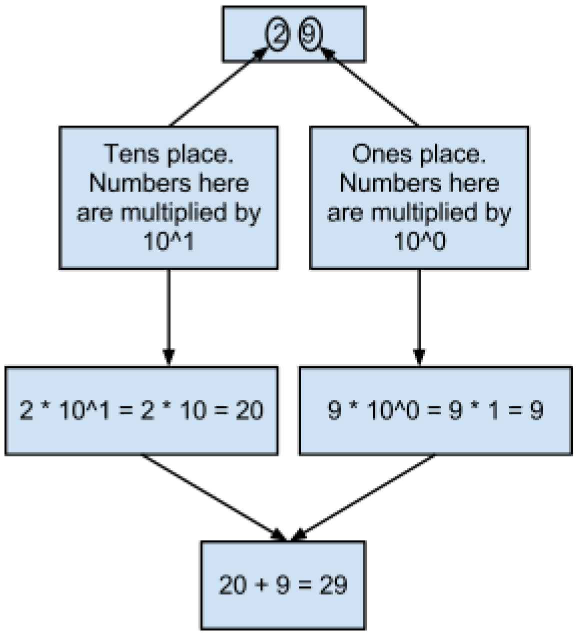

29
Tens place. Numbers here are multiplied by 10^1
Ones place. Numbers here are multiplied by 10^0
2 * 10^1 = 2 * 10 = 20
9 * 10^0 = 9 * 1 = 9
20 + 9 = 29

Figure 1 - Base-10 representation of 29

In base-10, each place value is a factor of ten: the ones place is 10^0, the tens place is 10^1, the hundreds place is 10^2, etc. Each number in each one of the places in base-10 notation is multiplied by the corresponding factor of ten. So the literal value of the number 29 is 2 values in the tenth place plus 9 values in the ones place, 2 * 10^1 + 9 * 10^0, coming out to 20 + 9 = 29.

In base-2, each place value is a factor of two, as opposed to ten: the “ones” place is 2^0, the “tens” place is 2^1, etc. So the value of 29 in binary (base-2) is

Figure 2 - Base-2 (binary) representation of 29

Confused? That’s okay, because we’re going to walk through some bits of this number! Let’s start with the right-most number, the least significant digit. This is the equivalent of the ones place in base-10 formating, so each number here is going to be multiplied by 2^0, or 1.

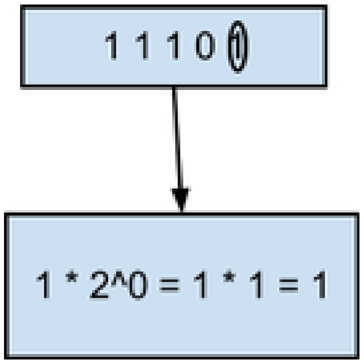


Figure 3 - Binary ones place

The next place is the equivalent of the tens place. Every value here is multiplied by 2^1, or 2.

Figure 4 - Binary tens place

The place after that is the equivalent of the hundreds place. Every value here is multiplied by 2^2, or 4.

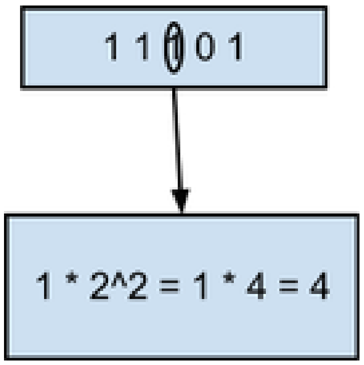


Figure 5 - Binary hundreds place

And on and on it goes. We won’t be going through every place in binary - we’ll let you try that for yourself. Here is the end result of the representation of binary.

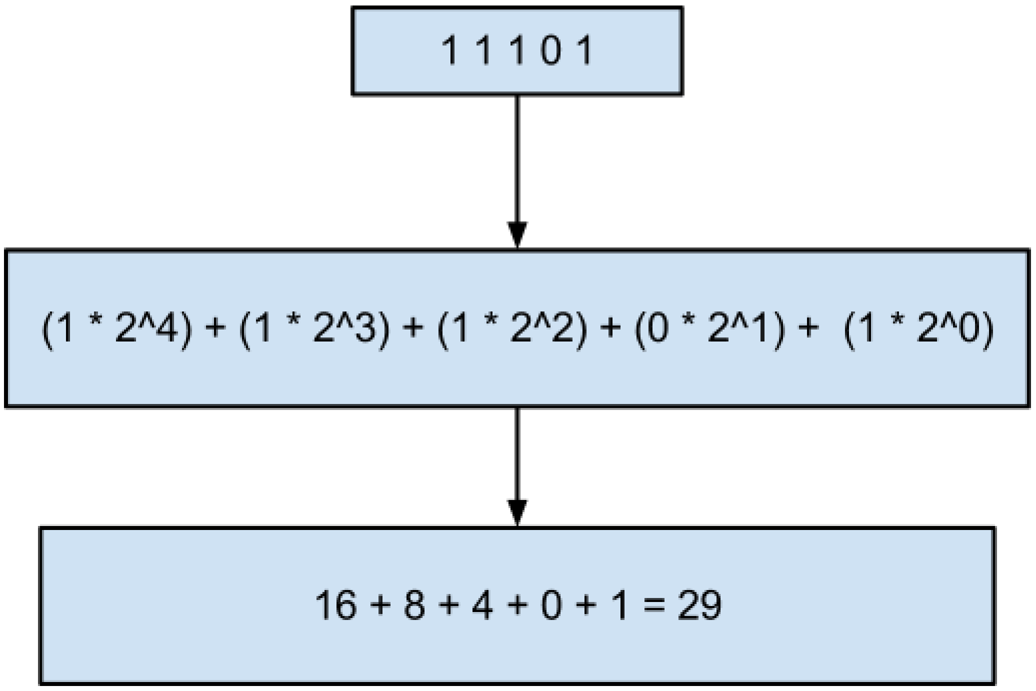


Figure 6 - 29 in binary

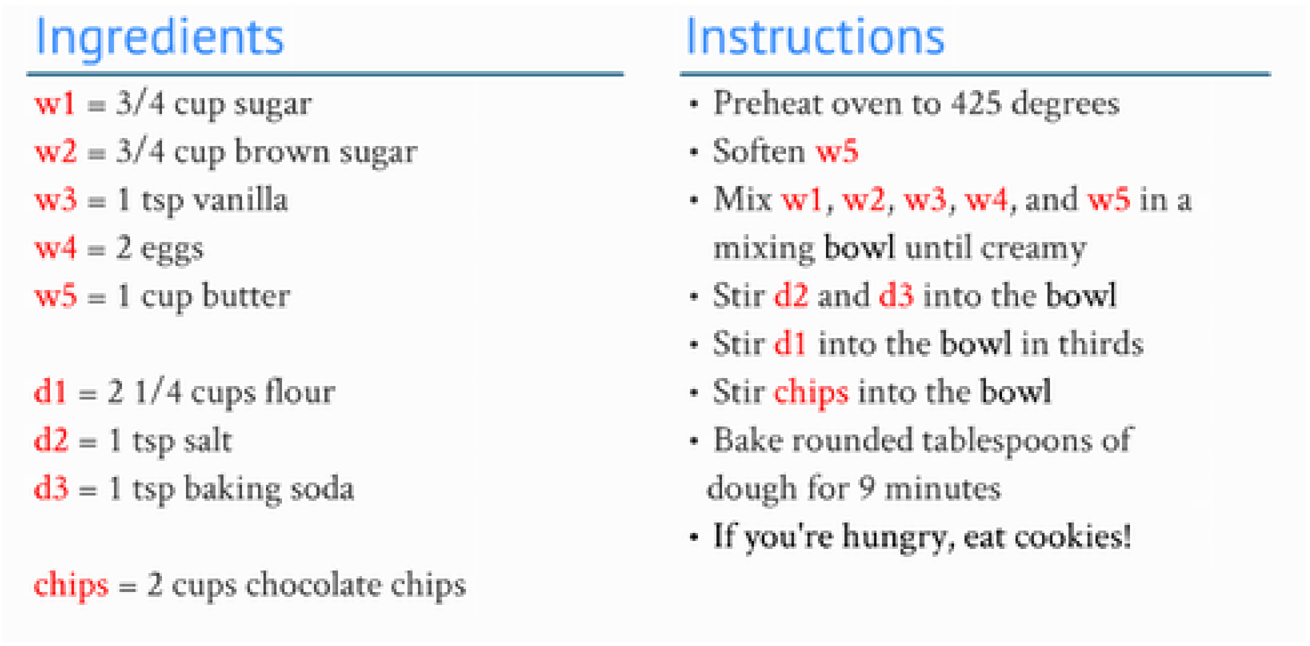
See? The same number in a totally different format. This simple numbering system is how computers view their world. Everything to them is either a zero or a one. If you're having trouble wrapping your mind around this concept, make sure you understand the second box in Figure 6. That shows how the value 29 is gleaned from the number 11101, starting with the least significant digit and ending with the most significant digit, multiplying each number by its corresponding place value.

Algorithms

What is an algorithm? It seems like one of those fancy words you would hear in a Hollywood movie when they try to explain how our superhero is able to hack the mainframe and save the day, but they don't have to be that complicated. In fact, algorithms are something we use every day! Let's start with a definition

Algorithm: a list of instructions for accomplishing a task that may be executed by a mechanism.

Pretty simple, right? You execute algorithms every time you bake cookies, or take the derivative of an equation, or maybe even when you're going through your morning routine. They are nothing more than a list of instructions that something - a human or a computer - goes through to accomplish a goal. Here's an example of a simple cooking recipe:



Ingredients

w1 = 3/4 cup sugar
w2 = 3/4 cup brown sugar
w3 = 1 tsp vanilla
w4 = 2 eggs
w5 = 1 cup butter

d1 = 2 1/4 cups flour
d2 = 1 tsp salt
d3 = 1 tsp baking soda

chips = 2 cups chocolate chips

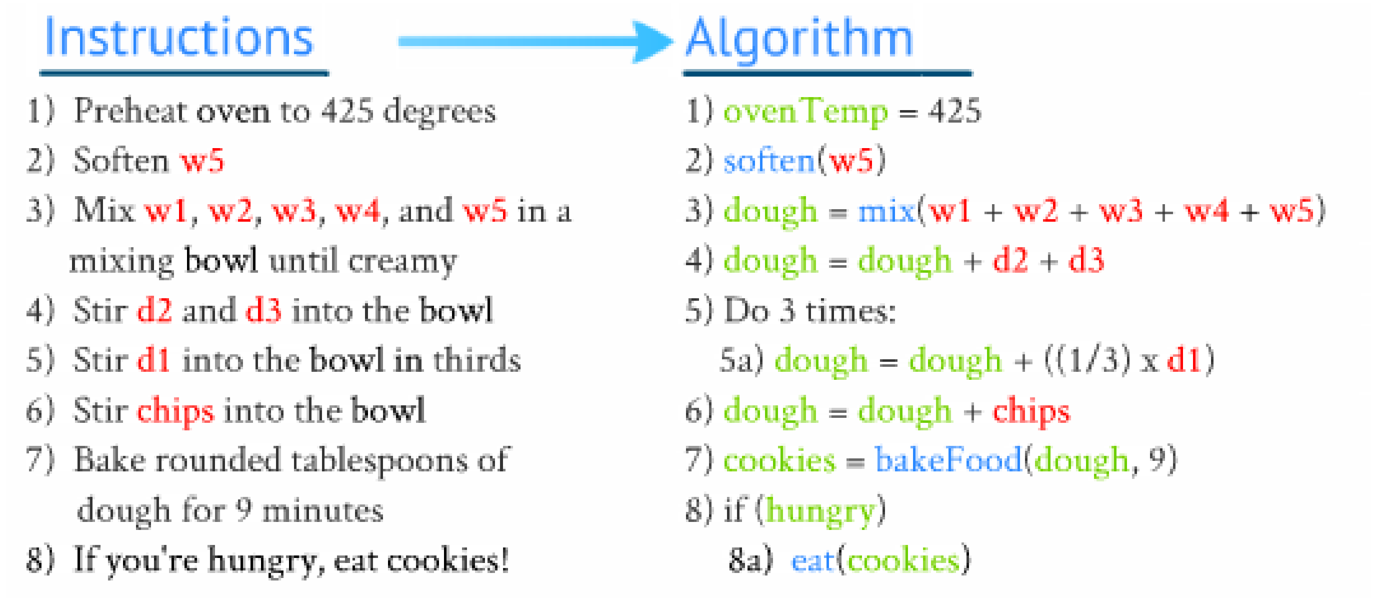
Instructions

- Preheat oven to 425 degrees
- Soften w5
- Mix w1, w2, w3, w4, and w5 in a mixing bowl until creamy
- Stir d2 and d3 into the bowl
- Stir d1 into the bowl in thirds
- Stir chips into the bowl
- Bake rounded tablespoons of dough for 9 minutes
- If you're hungry, eat cookies!

Figure 7 - A simple list of cooking instructions[1]

[1] Taken from Jeff Ringenberg's EECS 101 lecture, as are the rest of the pictures in the Algorithm portion

The instructions on the right side is the list of instructions that will be executed, the backbone of what an algorithm is. And here are those instructions turned into an algorithm:



Instructions	Algorithm
1) Preheat oven to 425 degrees	1) ovenTemp = 425
2) Soften w5	2) soften(w5)
3) Mix w1, w2, w3, w4, and w5 in a mixing bowl until creamy	3) dough = mix(w1 + w2 + w3 + w4 + w5)
4) Stir d2 and d3 into the bowl	4) dough = dough + d2 + d3
5) Stir d1 into the bowl in thirds	5) Do 3 times: 5a) dough = dough + ((1/3) x d1)
6) Stir chips into the bowl	6) dough = dough + chips
7) Bake rounded tablespoons of dough for 9 minutes	7) cookies = bakeFood(dough, 9)
8) If you're hungry, eat cookies!	8) if (hungry) 8a) eat(cookies)

Figure 8 - Cooking instructions turned into an algorithm

See? Baking cookies can be made into an algorithm. The instructions on the right side of Figure 8 might seem a little more complicated, but they're just the same as the instructions on the left side. They are now just written in psudo-code, which is a way to write out the basic steps of the algorithm using conventions that are similar to computer code.

Algorithms can be as easy as cooking or as complicated as Netflix recommending a set of titles based on your viewing of *Mean Girls*, but they all have three basic tools that they can implement in code to get their algorithm to run smoothly.

The first is that algorithms **execute in sequence**, the same way every time. The algorithm in Figure 8, if unchanged for the rest of time, will execute the same 8 steps in the same order, the same way, every time. That's why computers are so useful. Humans make mistakes, but computers don't, so you can trust that a computer will execute instructions in the same order every single time.

The second tool commonly used in algorithms is **selection**, which is choosing what set of instructions to do based upon some existing condition. For example, in our cooking algorithm, if we want to make sure we account for people who are lactose intolerant, we would have a statement that accounts for such, like the following

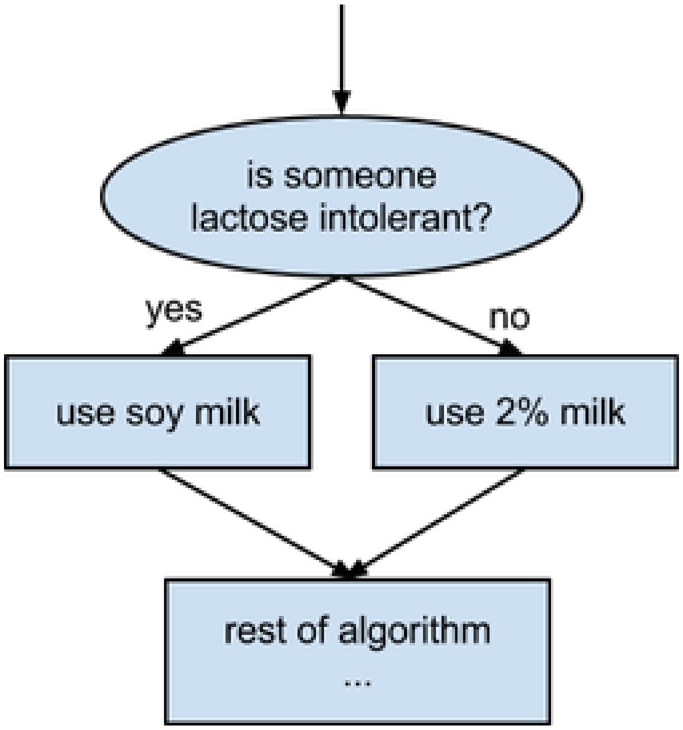
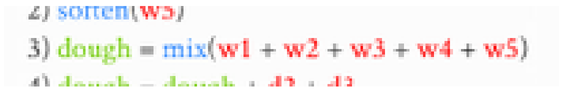


Figure 9 - selection statement

Figure 9 shows how selection works. If someone is lactose intolerant, a different set of instructions (using soy milk) is executed than if someone is not lactose intolerant (using 2% milk).

The third tool algorithms very often use is **iteration**, which is the executing of the same instructions multiple times. Why would you need to execute something multiple times? Let's think of the cooking recipe. You have to crack 3 eggs to make your cookies, so you crack one at a time, doing more or less the same action for every egg you use. So you are iterating over the same set of instructions for each one of the eggs. It is also depicted in Figure 8 in step 5, where the chef is supposed to stir the dough into three separate parts.

Now that you know the basics of an algorithm, lets look at something a little more complicated we're going to call **procedural abstraction**. Sounds pretty serious, huh? Well, it's not as hard as we decided to make it sound. Check out step 3 from our cooking algorithm below:



```
2) soften(w5)
3) dough = mix(w1 + w2 + w3 + w4 + w5)
[illegible]
```

Figure 10 - Procedural Abstraction

In order to get the dough, we're doing this weird thing called "mix" using all our ingredients. Now, it's pretty obvious that this "mix" thing mixes the ingredients we have up in order to create the dough. But what does "mix" do? And why does it look weird like that? Well, this is that procedural abstraction concept we were talking about earlier.

Procedural abstraction is a concept that programmers use to make computer code easier to read for a human. Instead of having step 3 happen in one step, we could have listed out all the steps required to make the dough. That would have been ugly though, and mixing the ingredients together is like an algorithm itself, right? So we take all these steps, put it in what's called a "**function**", and call the function when we want to execute the instructions in the code. Using these functions calls, we are able to make our algorithm easier to read and follow for anyone who wants to use the instructions. In our case above, the function is called mix, and the instructions are executed within that function.

Common Algorithms

Algorithms are so vital to computer science that many common types of algorithms have been developed for specific uses within the realm of computer science, the most common being sorting and search algorithms. Some common ones -- **Bubble Sort**, **Insertion Sort**, and **Binary Search** -- will be analyzed and reviewed in this study guide.

In addition to studying *what* these algorithms do, it is also important to study *how* these algorithms do what they do. Oftentimes in computer science, there exist multiple solutions to a singular problems. A common way to distinguish the good solutions from the bad is to look at the complexity of an algorithm, that is, how long it will take for the algorithm to finish. The faster the algorithm takes to produce a desired result, the better regarded the algorithm is. In order to measure the speed of the algorithm, computer scientists use **Big O notation**.

Big O Notation

Big O notation measures the amount of steps an algorithm takes to accomplish its goal, using the variable "n" to dictate how many steps it will take given an input of length "n" products. So a sorting algorithm's Big O notation is based off how quickly the algorithm can sort a list or array of products of size n. If we've created an algorithm that only takes one step per object in the input, the Big O notation for the algorithm would be $1 * n$, or

O(n)

For reasons that you will not have to worry about yet in your computer science development, we will ignore any constants multiplied by a value of n in Big O notation. So an algorithm that takes 3 steps per object in the input to complete will have the same input as an algorithm that takes 10 steps to complete: O(n).

Here is a list of common Big O notations in order from fastest run time to slowest.

Notation	Name
$O(1)$	Constant

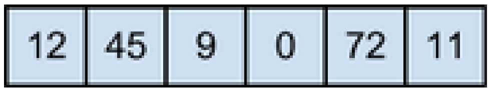
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n * \log n)$	N-log n
$O(n \text{^} 2)$	Quadratic
$O(n \text{^} c),\ c > 1$	Polynomial
O(n!)	Factorial

Ordered Big O Notation[2]

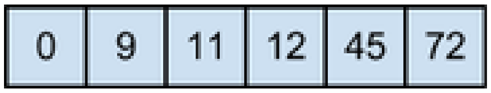
Now that we know some of the basics of studying algorithms, let's look at some specific ones.

Bubble Sort

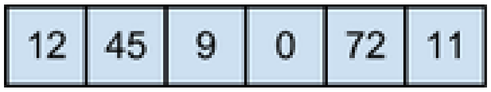
As its name implies, bubble sort is a sorting algorithm. So bubble sort is able to take an array of input like this



and turn it into this (if the algorithm is implemented to sort in ascending order)



Let's walk through how it does this. The basic definition of bubble sort is that you walk through the array, swapping adjacent items until you can iterate through the array without needing to swap any adjacent items. We'll do this step by step. Let's start with the input:

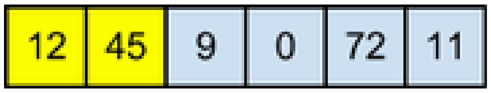


We pass through the array the first time, swapping two items when the item on the right is a smaller number than the object on the left. We will iterate through the entire array once.

Step 1

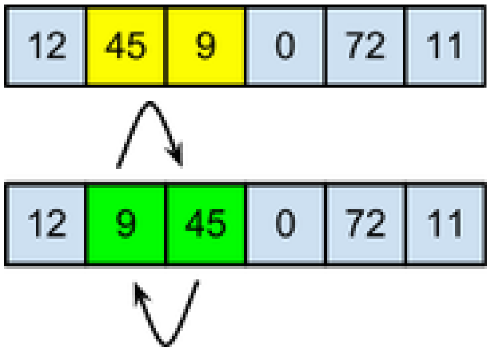
Step 1.1: Compare 12 and 45. They are in correct order: 12 is smaller than 45. Continue without doing anything.

[2] http://en.wikipedia.org/wiki/Big_O_notation



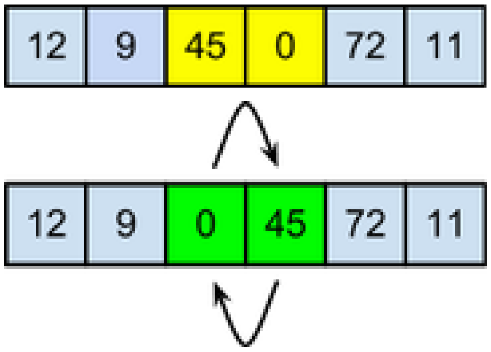
Step 1.1

Step 1.2: Compare 45 and 9. They are in the opposite order, so let’s swap the values.



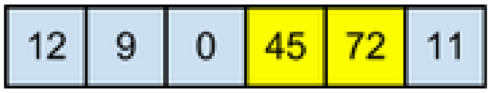
Step 1.2

Step 1.3: Compare 45 and 0. Also out of order, so let’s swap the values



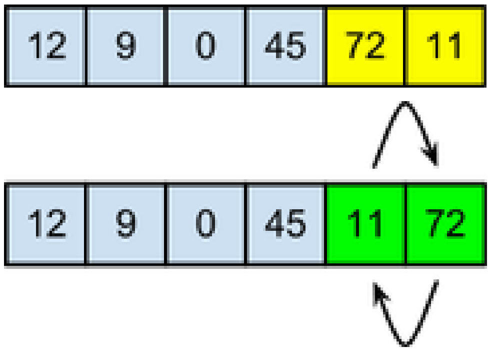
Step 1.3

Step 1.4: Compare 45 and 72. They’re in the right order, so don’t do anything.



Step 1.4

Step 1.5: Compare 72 and 11. They’re out of order, swap the values.



Step 1.5

At the end of step one, our array looks like this:

12	9	0	45	11	72

End of step 1

We will not go through each step one object at a time, because this is an opportunity for you to walk through the rest of the steps on your own to further learn the material. We will, however, list the end of each step, so you can check your progress as you step through the algorithm.

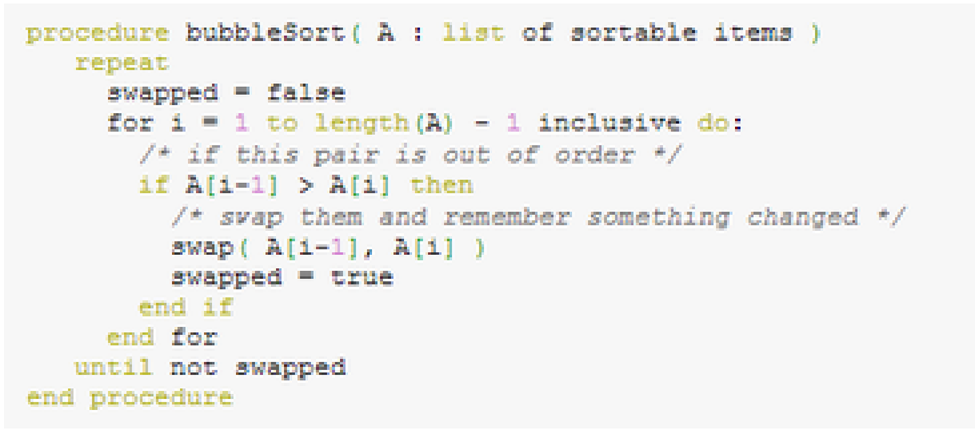
9	0	12	11	45	72

End of step 2

0	9	11	12	45	72

End of step 3

Depending on how you implement the algorithm in code, there may be another step to check if there are any more changes that you have to do. Once that check determines there are no more values to swap, the algorithm is done. Here is a pseudocode implementation of such a bubble sort



```
procedure bubbleSort( A : list of sortable items )
   repeat
     swapped = false
     for i = 1 to length(A) - 1 inclusive do:
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

Figure 11 - Bubble sort pseudocode[3]

Now, let's figure out the complexity of the algorithm using the Big O notation that we talked about earlier. There are three main measures of Big O notation that are generally attached to algorithms: best case, worst case, and average case scenarios. We'll start with the best case scenario.

[3] Taken from Wikipedia: http://en.wikipedia.org/wiki/Bubble_sort

When trying to figure out the best case Big O notation, all you need to do is think about what the best case scenario for a given algorithm is. What's the best case scenario for a sorting algorithm? When the algorithm doesn't have to do anything, the array is already sorted! The array in the from of `0 1 2 3 4 5` is already sorted. The only thing the algorithm needs to do is iterate through the array once, making n number of comparisons. So the big O notation for best case bubble sort is

$$O(n)$$

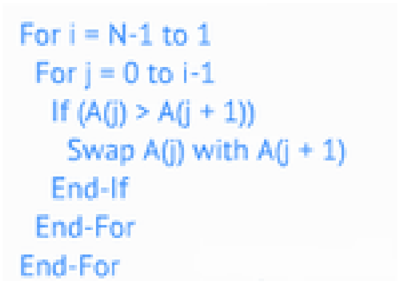
Best case - bubble sort

Now, what's the worst case for a sorting algorithm? If an array is in a complete opposite order than what is desired, the algorithm has to do the most amount of iterations in order to get the items in the best order. If it's not completely clear, go step by step and sort the array `0 1 2 3 4 5` in descending order, and see how many steps it takes you. In the end, you will see that you will need to iterate through the array n times in order to move the first element in the array to the other n, needing to perform n steps every time, making the complexity of the worst case bubble sort

$$O(n * n) = O(n^2)$$

Worst case - bubble sort

This is a full magnitude of n worst than the best case scenario of bubble sort. It helps if you think about worst case scenario in comparison to how the code is written. Bubble sort needs multiple, nested loops (one loop used within another loop) in order to iterate through the array twice to completely sort the array. The outer loop takes n iterations to complete, each step completing an entire sweep through the array, which requires n steps each to complete. This simplified version of bubble sort can be seen in this psuedocode depiction of the algorithm (note the two "for loops" at the beginning of the code)



```
For i = N-1 to 1
  For j = 0 to i-1
    If (A(j) > A(j + 1))
      Swap A(j) with A(j + 1)
    End-If
  End-For
End-For
```

Figure 12 - Simple bubble sort pseudo code[4]

Last, we will look at the average case scenario. Since this can be a variety of things (unlike best and worst case), many times average case for an algorithm is estimated based on general knowledge of how the algorithm is laid out and also what its best and worst case scenarios are. With bubble sort, as we saw above, we know that the two loops are required to account for the worst case scenario. So the average case can't be worst than $O(n^2)$ complexity, but it can't be

[4] Jeff Ringenberg Lecture

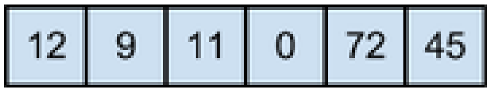
better than the best case scenario of $O(n)$. According to the table above that lists out the order of Big O notations, the only one between the two $O(n)$ and $O(n\hat{\ }2)$ is $O(n * \log n)$. You'll have to take my word for it (and if you choose to pursue Computer Science you will learn the reason why eventually) but an algorithm set up in the same manner as bubble sort -- nested "for" loops -- cannot have a complexity related to log n. Any time you see multiple loops in an algorithm, the average complexity is usually in the same range as $O(n\hat{\ }2)$, and since average case is an estimate as opposed to a real representation, we can safely say that the average complexity of bubble sort is

$$O(n\hat{\ }2)$$

Average case - bubble sort

Insertion Sort

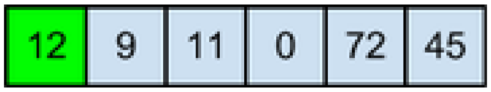
The second sorting algorithm we will look at is insertion sort. A little more complex than bubble sort, insertion sort goes through an array, picking up a value and moving it to its correct spot in the array, slowly building a sorted array as the algorithm progresses. I've found that insertion sort is better explained in picture than words, so we'll just get straight into the example. We'll work with this new array



At each step in the array, you add another element to your sorted array until you have a completely sorted array. So, the first part of the first step of this example is redundant and unnecessary, but is done to prove a point.

Step 1

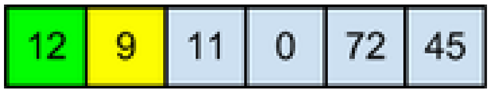
Check the leftmost element (12) that has not been put in the sorted array. It's already in order (since its the only member of the sorted array), so we don't need to do anything.



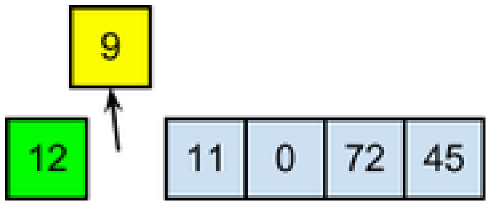
We will mark the already-sorted elements in green.

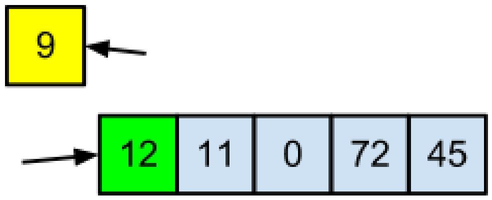
Step 2

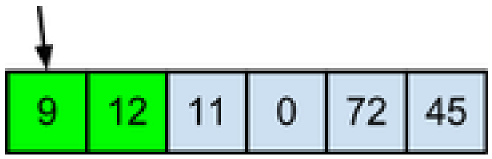
Check the leftmost unsorted element (9).



Compare it with all the elements of the unsorted array until you find a place to put it. Since it's so early on in the algorithm, we only have one element in the array, so we just place it on the other side of 12



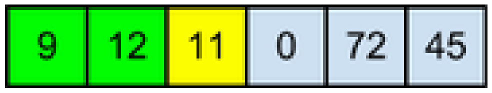


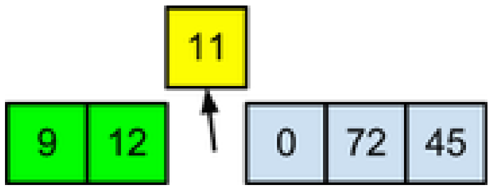


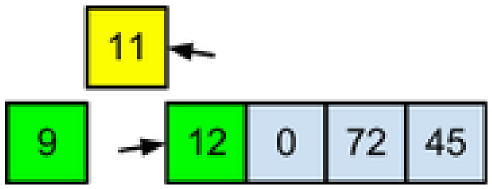
We now have two sorted elements.

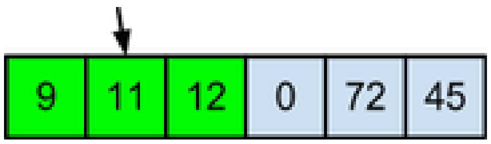
Step 3

11 is now the left-most un-sorted element. We need to place it in the right order in the sorted array, so we compare it to all the values of the currently sorted array. We see that it fits in between 9 and 12 in the sorted array.



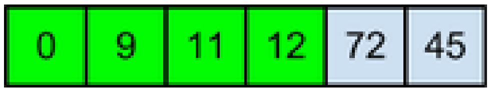




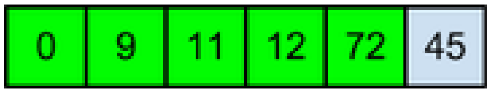


We now have three values in our sorted array.

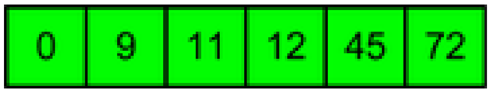
For each one of the steps, we compare the value of the next unsorted element with all the values of the sorted elements in order to find the best spot for it. You can work out the rest of the steps one by one, the results are displayed below.



Step 4

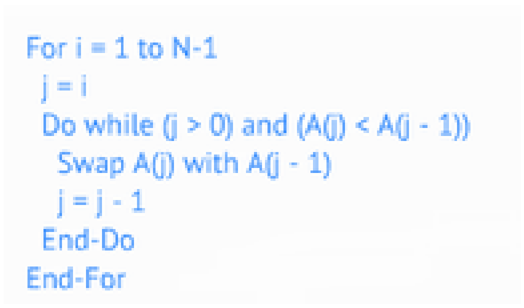


Step 5



Step 6

So for every step of insertion sort, we increase the size of the sorted array with which we do the comparisons. Given what we learned in bubble sort, how do you think the algorithm is set up? If you guessed “nested loops again!” then you guessed right. Here is a pseudocode implementation of insertion sort.



```
For i = 1 to N-1
  j = i
  Do while (j > 0) and (A(j) < A(j - 1))
    Swap A(j) with A(j - 1)
    j = j - 1
  End-Do
End-For
```

Figure 13 - Insertion Sort[5]

As you can see, there are two loops the algorithm uses to complete the sort: one “for” loop and one “do-while” loop[6]. Using this information, can you guess the average case for insertion sort? When we looked at two nested loops within an algorithm previously, we saw that the average

[5] Jeff Ringenberg Lecture

[6] http://en.wikipedia.org/wiki/Do_while_loop

case was $O(n^2)$. The same goes for this algorithm as well:

$$O(n^2)$$

Average case - Insertion Sort

Can you think of what the worst case scenario could be? We can do this by analyzing the code like we did with average case. Since there are two nested loops, the worst case scenario is that we iterate through the two loops completely, making the worst case Big O notation for insertion sort

$$O(n^2)$$

Worst case - Insertion Sort

Now what about best case? Insertion sort is a little different than bubble sort, because you can break down the complexity in multiple ways: the number of comparisons and the number of swaps. The best case scenario for this sorting algorithm is when no numbers are swapped, so the Big O notation for swaps is constant, or $O(1)$. However, best case scenario still makes one comparison per value, determining for each value in the array that they are already in the correct position; this makes the Big O notation for comparisons $O(n)$.

Combining the two, we have the complexity of the best case scenario for insertion sort being $O(n) + O(1)$. When representing multiple parts of an algorithm that contribute to an overall Big O notation like we have here, we let the largest value determine the overall Big O notation of the algorithm. In this case, $O(n)$ is larger than $O(1)$, so we say that the best case Big O notation of Insertion sort is

$$O(n)$$

Best case - Insertion Sort.

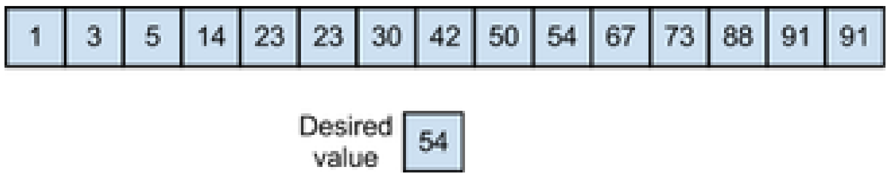
Binary Search

We've looked at only one type of algorithm in detail so far: the sorting algorithm. We will now look at another simple type of algorithm known as a search algorithm. Search algorithms work best with pre-sorted input, and the only way this particular search algorithm works is if it assumes that its input is already sorted. If the values are not sorted, the only way to find an object is to loop through the entire input array to find a value. However, when we deal with sorted arrays, we can develop clever tricks to complete the algorithm more efficiently.

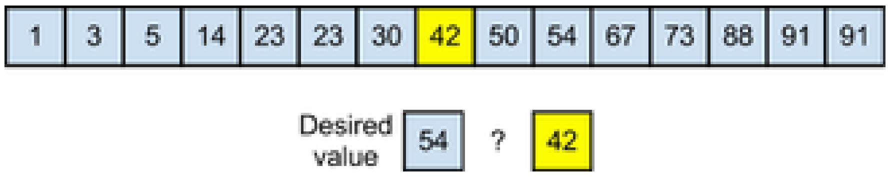
A binary search has two inputs: a sorted array (in this example assumed sorted in ascending order), and the value that we are looking for within the sorted array. In order to find the value, the algorithm takes the middle value of the array and compares it to the value that we are looking for. There are three possibilities that we must check:

1. If the middle value is less than the desired value
2. If the middle value is greater than the desired value
3. If the middle value is equal to the desired value

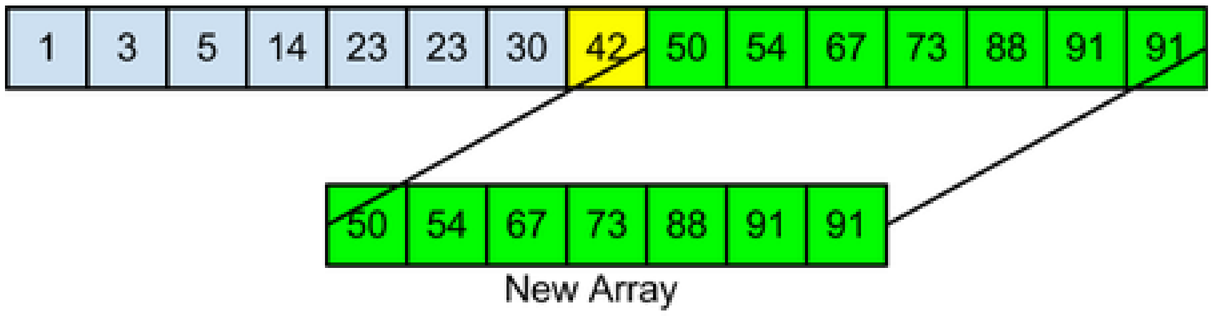
If we get case 1, we know that the value, if it exists, is in the right half of the array. If we get case 2, we know that the value, if it exists, is in the left half of the array. If we get case 3, we are done with the search, while cases 1 and 2 require the algorithm to keep looking for the value. To do so, it takes the half of the array that the desired value could be in (the right half in case 1, the left half in case 2) and does the same exact steps using the smaller array. This uses a really awesome concept that we won't get into called recursion[7]. Let's walk through the algorithm step by step, just so we can be perfectly clear. Here is our sorted input:



Let's start the search! We start with the middle value of the array, and compare it with the desired value.

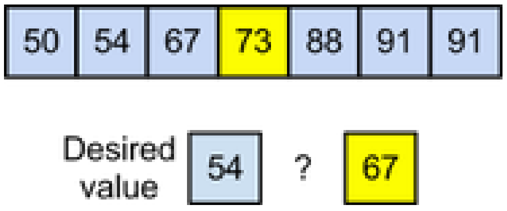


We clearly see that the desired value (54) is greater than the middle value of our current array, case 2 from above. Since this is case two, we take the right side of the array and use those values to continue the search

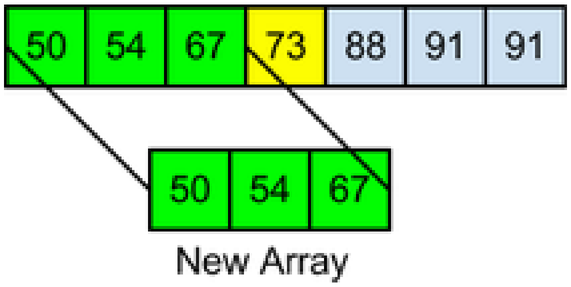


Using this new array, we can continue to the next step. We will do everything we did in the first step, just with a smaller array. We start by comparing the middle value of our new input.

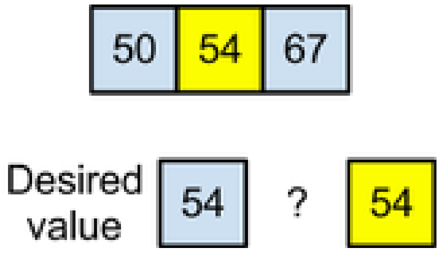
[7] If you're curious, you can check out its awesomeness here.



We see that our desired value is less than the middle value, so this iteration of the binary search algorithm has produced case 2. We have to take the left side of the array and use that as our input now.



Since we still don't have our value, let's continue the binary search using our new array. Compare the middle value again!



Hey, look at that, we found our value. We have now hit the third possibility of an outcome of an iteration of binary search: actually finding our value. We are now done.

Desired value
54
==
54

Let's look at the complexity for this specific algorithm now. Given an input of 15 values, it took our search algorithm 3 compares to find the right answer. Those are pretty good numbers, seeing as it took less than n comparisons to finish. The sorting algorithms we used couldn't get lower than n if they tried. The name for this kind of complexity in an algorithm is **logarithmic** complexity, denoted by $O(\log n)$. What this value really comes out to is roughly the base-2 log of n. In this case, the $O(\log n)$ of this search algorithm is log(15) / log(2). (This comes out to be more than the actual number, but our number we found above did not account for the resizing of the arrays). We have just found the average case performance of binary search (which also happens to be the worst case performance as well)

$$O(\log n)$$

Worst/average case - Binary Search

Now what is the best possible scenario for binary search? If 54 had been the first value we compared with our desired value, we would have been done with the search after the first step. This is the best case scenario -- constant lookup time

$$O(1)$$

Best case - Binary Search

Data Storage

You've learned a lot about computers so far. You've learned their language - how they represent everything in binary - and you've learned how they execute various things - using algorithms. But where is all this binary data stored? They need some physical media on which to store all this stuff!

In the early days of computers, all this stuff computers needed was stored on punch cards and physical tapes (similar to the material used in cassette tapes). All of the zeros and ones that computers needed to know was stored on this physical media. As computers got more complex and needed to get data faster and more conveniently, data storage got more and more complex, going from floppy disks to optical drives (including CDs and hard drives) to solid-state storage (remember flash drives?). As computers have evolved, so has their storage.

Now that we know *what* the data can be stored on, we can now talk about *how* the information is stored. There are countless ways to answer the *how* question; your Facebook friends are stored in a different manner than the essay you have stored on your desktop that you haven't quite decided to finish yet. We will focus on one type of storage known as **databases**.

Databases

The word "database" may seem intimidating, but the word serves as the definition to itself. A database is really just a base (or a collection) of data, a collection of data stored a location. You interact a variety of databases all the time, whether it's online, on your laptop, or on your phone. Everyone uses databases, but not all databases are created equal.

One specific type of database, one with which you might interact every day, is called a relational database, which are in use in almost every website you can go to. A relational database simplifies storage, putting everything in a different collection of objects called a **table**. The best way to explain tables is to show one. Check out this example:

Player Name	Position	Team
Steve Smith	Wide Receiver	Carolina Panthers

Peyton Manning	Quarterback	Denver Broncos
Steve Smith	Wide Receiver	St. Louis Rams
Arian Foster	Running Back	Houston Texans

Figure 14- Relational Database (table) of NFL players

Tables have a couple of defining characteristics. Each table is made up of one or more columns, each with its own header defining what is in the column. In the example above we have three columns whose headers are **Player Name**, **Position**, and **Team**. Each **entry** into this table has a player name, position, and a team.

Another defining characteristic of relational databases is the use of **primary keys**, which enforce that every entry within a table has a value that sets it apart from the others so no two entries have the same primary key. In a table of American citizens, a primary key that would be unique to each entry could be a person's Social Security number.

Now check out that table in Figure 14 again. Can we rely on any of the existing columns to uniquely identify the players as our primary key? We have repeats in both **Player Name** and **Position**, so we can't use those as our primary key, there are repeats in both columns! We could use **Team**, as none of the four players are on the same team. But what if we add Arian Foster's teammate Andre Johnson to the table? We will have two players from the Houston Texans on the team. So we can't rely on **Team**.

Sometimes when making a database you have to make your own primary key, something as simple as a number. Social security numbers are really just complex primary keys to identify people. So what if we gave each one of our players in our database a unique number that can correspond only to them? It would look something like this:

Player ID (primary key)	**Player Name**	**Position**	**Team**
1	Steve Smith	Wide Receiver	Carolina Panthers
2	Peyton Manning	Quarterback	Denver Broncos
3	Steve Smith	Wide Receiver	St. Louis Rams
4	Arian Foster	Running Back	Houston Texans
5	Andre Johnson	Wide Receiver	Houston Texans

Figure 15 - A valid relational database table

Look at that. Now we can add any player we want to the table, and we will be able to uniquely identify them by their **Player ID**. This is now a **valid** relational database table: each entry has a primary key that uniquely identifies the entry.

Relational databases are very useful in storing information that needs to be looked up as fast as possible. You can have multiple tables within one database as well, making it easier for lookups to occur. Using our football player example, we can have a table that has stats for each player, and a separate table that stores basic information like the one in Figure 15, linking them both by the primary key of a player.

Cloud Computing

The term “cloud computing” was coined in order to find a way to refer to stuff that a user interacts with online. It’s an easy term that is used to refer to a variety of different storage methods (sometimes relational databases) that are stored on what are called **servers**. Servers are just computers that are dedicated to running only to serve the needs of users on the network. They’re not like a personal computer, such as a laptop or a desktop, that is readily available for human interaction. Servers are used to store websites, search results, login credentials, Google Documents, Netflix movies, and Amazon inventory.

Cloud computing defines a direction in which the use of servers has gone in recent years, leveraging the available mass online storage capacities that servers provide. Companies like Dropbox, Apple, and Box have all used online servers upon which users can store data, having it accessible at any time with a computer and an internet connection, vastly changing the face of computer storage. No longer do people have to carry around physical hardware -- in the form of flash drives and computers -- to have access to files. As more and more companies choose to go online with storage, the amount a user can have stored online and available at any point in time is almost limitless.

Deleting Data

We’ve learned before that everything in a computer at its lowest level is represented in zeros and ones. What we haven’t learned yet is how computers pull all these zeros and ones off of a hard drive to display it to the user. Similar to a table of contents in a book, a computer’s operating system has an index of all the files it has on its hard drive so that it knows exactly where to look when a user selects a given file. Think of it like an address book: if someone is looking for a particular person in a city, one solution could be to go knock on every single door, stopping only when you found what you were looking for. That doesn’t sound very fun or efficient, does it? If you do the same thing, but with an address book, you only need to look up which house to check and go and knock on one door.

The index that a computer has of all the files on disk works in the same way the address book helps someone look for a house. A computer *could* go through all the space it has on its hard drive and look for the file, but that would take so long that computers would almost be useless. As a result, they index all of the files they use so they can look up file locations immediately. (We can use Big O notation here! Look ups in an index can be done by computers in constant time: $O(1)$.)

Next time you delete something large from your computer, notice how quickly it is accomplished. Ever wonder why? As opposed to just clearing all of the actual data for a file off of a hard drive, the computer just deleted the entry in the index, removing it from its own address book. Thus, the computer is also able to know what space is free on its hard drive based upon what is listed in the index. If space on the hard drive is not listed in the index, then it will be safe to write to. This is pretty clever, and makes stuff move faster on a computer, but can you think of any drawbacks of this?

Since the operating system doesn't actually erase the data of the file off of the disk, the actual file will exist on disk until it is overwritten with another file by the operating system. The only reason the computer has no idea the file is there is because the file is still in the index. However, if someone clever enough gets a hold of the hard drive, they will be able to recover the file as long as it hasn't been overwritten, exposing a security flaw in how computers delete data. The only way to ensure that the information you want deleted is *permanently* deleted is by wiping the data from disk, usually by writing all zeros to all the space where the file existed.

Security

As more and more people are using computers, security is becoming more of an important issue to software developers. There are mean people in the world whose only goal in life is to mess with other peoples stuff; they hack away, trying to expose vulnerabilities and figure out how to break into things. Sometimes times this can be beneficial; exposed security flaws help a software developer find bugs and fixes to those bugs. It has also given birth to a whole aspect of computer science that is focused on keeping your things safe while keeping other people out of your stuff.

Cryptography

Computer scientists didn't invent security; people have had the need to protect sensitive information for longer than computers have been around. Julius Caesar used simple ciphers in order to send secure messages to parts of his army, removing the risk of exposure if the messages would be intercepted[8]. Using a simple **cipher** -- a pair of algorithms that can be used to encrypt and decrypt data[9] -- he was able to send these messages, knowing with a good amount of confidence that the messages could not be used against them. This practice of using secure methods in order to communicate messages without risking exposure from a third party is known as **cryptography**.

[8] http://en.wikipedia.org/wiki/Caesar_cipher

[9] Jeff Ringenberg Lecture

Cryptography is all around computers. Anytime you go to a new website in your web browser, tons of cryptographic measures are taken to ensure that the data you are sending all across the internet is secure. This communication -- cryptography in general -- is broken up into two parts. Encryption converts the plaintext of the message that is being sent into an unreadable format we will call ciphertext. Decryption then takes this ciphertext and converts it back into plaintext so that the person who receives the message can also read the message.

One example of a basic cipher is a substitution cipher, a class of cipher into which the Caesar cipher falls. Substitution ciphers are not very secure, as they rely only on the substitution of one value with another. The Caesar cipher, for example, shifted letters by a numerical value in the alphabet. So a message that says

```
Hey Caesar
```

that is encrypted using a shift of 3 would say

```
Khb Fdhvdu
```

Each letter is shifted three spots in the alphabet. Look at the first letter: H, the 8th letter in the alphabet, is changed to 11th letter in the alphabet. This might not be that easy for a human to figure out, but with the advancement of computers substitution ciphers have been rendered virtually useless, as they are easily broken using basic algorithms.

Let's get into some currently-implemented examples of cryptography. Not all cryptographic techniques are created equal. Like many things in computer science, the end goal -- in this case, secure communication -- can be reached using different but similar methods. Both of the techniques reviewed in this study guide will use a combination of ciphers and **keys** -- values input into the cipher algorithm(s) to encrypt and decrypt the data. Both ensure secure communication, going about it in two different ways.

Symmetric Key Cryptography

Symmetric key cryptography -- sometimes known as Diffie-Hellman key exchange -- is a very math-intensive process. It can be compared to a numerical padlock in theory, being a procedure that is easy to lock but difficult to unlock. Given an infinite amount of time, any padlock can be broken, it will just take a long while to do so. It's the same with symmetric key cryptography; the process is vital to attacks, but because of its use of mathematical concepts it is very difficult to figure out.

The basis of this technique is in its use of modular arithmetic[10]. Instead of getting into the detail of the math, watch this YouTube video to get a better understanding of the math involved in the process. Its about 5 minutes, and the use of images works better than explaining it with words.

[10] http://en.wikipedia.org/wiki/Modular_arithmetic

The important takeaway from symmetric key cryptography is that it is a process that, though very secure, can be broken if given enough time. Currently, the processing power of computers cannot yet overcome the mathematical backbone of this cryptographic process, but there's no telling what computers will be able to do in the future. The process is one that is easy to lock and difficult to unlock if you don't know the code with which to unlock it.

Public Key Cryptography

The idea of public key cryptography centers around the use of keys and ciphers. Each party has two keys: one public key, which anyone in the world can know, and one private key, which should be kept secret. As long as no one knows a party's private key, the process is secure. The cool thing about these keys is that they are used to encrypt and decrypt one another: messages encrypted with a party's private key can only be decrypted with that same party's public key and vice versa.

The process is set up using two steps, what we will call encrypting and signing the message. In order to send a secure message, you will need to guarantee that these two steps are accomplished. Encrypting the message it is important because you want to encrypt it so that only the party to whom you are sending the message can decrypt the message. Signing the message is important because you want the person who receives the message to know for sure that the message came from you. All this can be done using one cipher algorithm and four keys.

Let's use an example to explain the process -- I've found that much more useful when explaining public key encryption. We have to parties involved in communication: Party A and Party B. Since each party has two keys it uses, we have four keys: PrivateA, PublicA, PrivateB, and PublicB. Let's look at how these keys can be used to help parties A and B securely communicate with one another.

Step 1 - Encryption

Party A wants to encrypt the message so only Party B can decrypt the message. To ensure this, you need to make sure the message can be decrypted using Party B's private key -- the key that only Party B knows -- guaranteeing the only people who can decrypt the message is Party B. But how can we do this?

Recall before that I mentioned that public and private keys complement one another. One party's public key can encrypt a message that can only be decrypted by that same party's private key. Since everyone knows Party B's public key -- the defining characteristic of public keys -- anyone can encrypt a message that only Party B can read. All Party A has to do is take Party B's public key and encrypt its message using that,

Step 2 - Signing

But how can Party B ensure that Party A was the one who sent the message? There needs to be some type of electronic signature that Party A can put on the message in order to tell Party B that they were indeed the one to send the message. But how?

Recall that a party's private key is/should only be known by the party itself, and messages that are encrypted using this key can be decrypted using the same party's public key. So once the message has been encrypted, Party A can then take the encrypted message and encrypt it one more time using its private key. That way, the only way to decrypt it is to use Party A's public key, and Party B can be sure that Party A sent the message, as it was encrypted using something only Party A knows.

Cryptanalysis

Just as understanding cryptographic processes is important, understanding how to break these processes is important as well. That's the only way to be certain that what you're doing is unbreakable. We will just go over some basic cryptanalysis techniques here to get you familiar with the process.

Frequency Analysis

Frequency analysis only works when confronted with a substitution cipher. The concept of frequency analysis takes the occurrence rates of letters or strings of letters and tries to figure out what the most common occurrences are. This technique only works if the ciphertext is long enough to notice enough of a pattern to ensure that the substitutions are correct.

Brute Force

The simplest form of cryptanalysis, it centers around using vast amounts of processing power to figure out a solution to the encryption. It tries every possible key that can be found, inserting each into the cipher to see if the plaintext is ever revealed.

Computer Hardware

We've talked a lot about software so far, but we have done little in the way of telling you how the software runs on the hardware on which it lives. We wouldn't have software without hardware, so it's important to know some basics of computer hardware to appreciate fully what we are capable of with these devices at our finger tips.

Digital Logic

At the start of this study guide we learned about bits and how they are used to represent the data in a computer. This concept of bit notation is manifested in hardware, as hardware is based upon binary logic. At the root of this logic are three basic logical operations: AND, OR, and NOT.

(All operations are of the form A [operation] B = C, where A, B and C are all single bits, and the value of C maps to true when C = 1 and false when C = 0)

AND

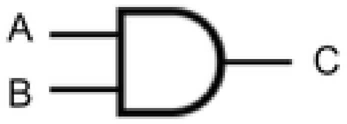


Figure 16 - An “AND” logic gate

The value of AND evaluates to true if and only if the two values input into the logic equation are both true, or equal to 1. If you look at the logic gate above, you see the two inputs A and B. If both of these inputs are 1, the output C will also be 1. In all other cases, C is false, or equal to 0. Let’s enumerate all of the possible combinations of A and B with their output value of C.

A	B	C (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

Figure 17 - AND truth table

What we just created is known in the world of digital logic as a **truth table**. A truth table shows the output of a given logic statement for all the possible inputs of data. As you can see in Figure 17, the only time C is true is when both A and B are true.

OR

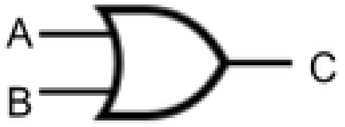


Figure 18 - An “OR” logic gate

The OR logic gate evaluates to true when either A or B are true. Here’s a look at its truth table.

A	B	C (A OR B)
0	0	0
0	1	1
1	0	1
1	1	1

Figure 18 - OR truth table

As you can see, the only time an OR gate does not evaluate to true is when either A or B (or both) are true.

NOT

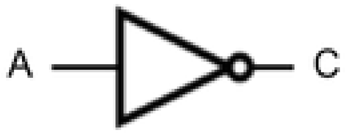


Figure 19 - A “NOT” logic gate.

Unlike AND and OR, a NOT gate only take in one value as input and outputs the opposite of that value. Here is the truth table, much simpler than the others

A	C (NOT A)
0	1
1	0

Figure 20 - NOT truth table.

These three basic operations lay the groundwork for all digital logic within a computer. Everything in computer hardware can be broken down to these three basic operations. Now, let’s look at some very common, more complex operations.

NAND

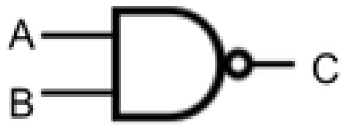
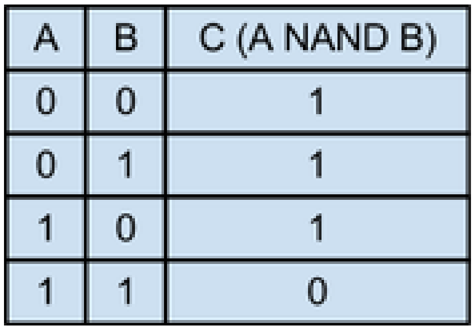


Figure 21 - NAND gate

NAND -- short for NOT AND -- is a simple operation that consists of one AND gate and one NOT gate. The result of the operation is just the opposite of what an AND gate spits out, resulting in an output of 1 only when either A or B evaluate to false. Here is the truth table.



A	B	C (A NAND B)
0	0	1
0	1	1
1	0	1
1	1	0

Figure 22 - NAND truth table

If you compare it to the AND truth table (Figure 17) you will see that each result of the NAND gate is just the opposite result of the AND gate.

NOR

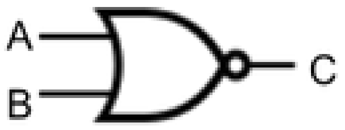
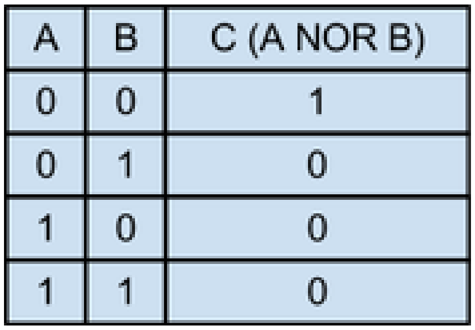


Figure 23 - NOR gate.

From the same family of logic operations as NAND, NOR -- short for NOT OR -- is just the opposite of an OR gate, evaluating to true only when neither A nor B are true. Here's the truth table. As you can see, it's the exact opposite of the truth table of OR.



A	B	C (A NOR B)
0	0	1
0	1	0
1	0	0
1	1	0

Figure 24 - NOR truth table

XOR

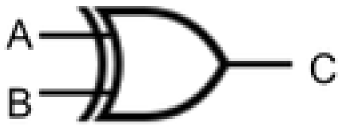
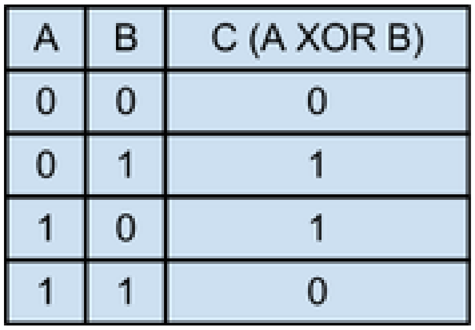


Figure 25 - XOR gate

XOR is a little more complicated than the other previously reviewed logic gates. XOR evaluates to true only when exactly one value, A or B, evaluates to true. Here is the truth table



A	B	C (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

Figure 26 - XOR truth table

A simplified way of writing A XOR B is (A AND (NOT B)) OR ((NOT A) AND B)

Now that we have gone over all of the most basic logic operations, we can move on to building some cool stuff using logic gates. We're going to build an Adder -- a series of logic gates that can solve binary addition. Before we do that, we have to get you acquainted with the concept of binary addition.

Binary Addition

Let's start by showing basic addition of bits, and how it can be used to add binary. Here are the four basic combinations of bits, and their results when they are added together

0 + 0 = 00
0 + 1 = 01
1 + 0 = 01
1 + 1 = 10
1 + 1 + 1 = 11

This may seem a little complicated, but lets remember that we're working with binary values. The first three are pretty straight forward: the results are 0 when nothing is added together, or 1 when 1 is added to zero. The second to last equation equation, 1 + 1, results in the answer 2, which in binary is displayed with the value 10. The last equation, 1 + 1 + 1, results in the answer 3, displayed in binary with the value 11.

Let's do an example of binary addition using some simple values: 11 and 7. As you know, 11 + 7 = 18. Now to do binary addition, we have to get both of our imputed values into binary representation.

11 → 1011
7 → 0111

Now let’s do some binary addition. You can do it by yourself, and check your results with the values below

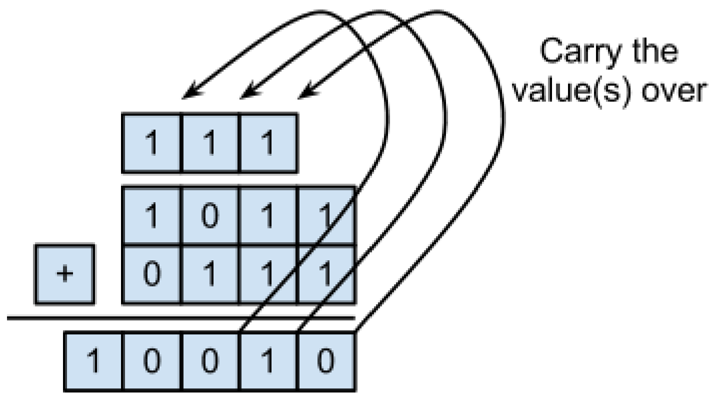


Figure 27 - Binary Addition

The resulting binary is 10010, which in decimal notation comes out to be 18. We successfully learned binary addition and applied this addition in an equation. Now let’s learn how we can make this using logic gates.

Adders

In order to make a fully functioning Adder using logic gates, we have to get the basic operation of bit plus bit addition settled. If you take a look at the additions of bits, we can see some patterns that upon which we can apply logic gates. While adding two values, one must be responsible for two outputs: the sum and the carry values. Lets make a truth table for each to help us out. Here is one that has what each value of the sum and the carry values are based upon a two-bit input.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 28 - Truth table of 2-bit addition

Let’s see if we can’t make these into logic gates. We’ll look at the results of Sum first; notice anything familiar? It eerily similar to the XOR truth table we saw in Figure 26, so we can say that the Sum of two inputs A and B is A XOR B.

Now let's look at the Carry value. This one should look familiar too -- its the truth table for AND. So we can say that the Carry of two inputs A and B is A AND B. Combining these, we can make use logic gates to create what we're going to call a Half Adder.

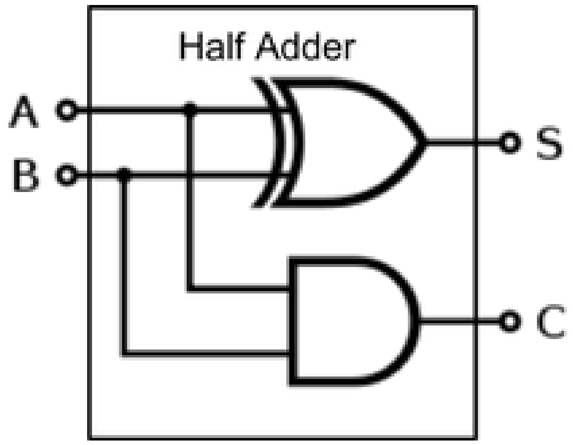


Figure 29 - Half Adder

This simple logic circuit has two inputs, A and B, with two outputs, Sum and Carry. This is helpful, but what about the carry values? It would work fine for the addition of the least significant digit, as there is no carry value to be added for that, but what about for the addition of the digits that can have carries? We'll need to make something we'll call a Full Adder

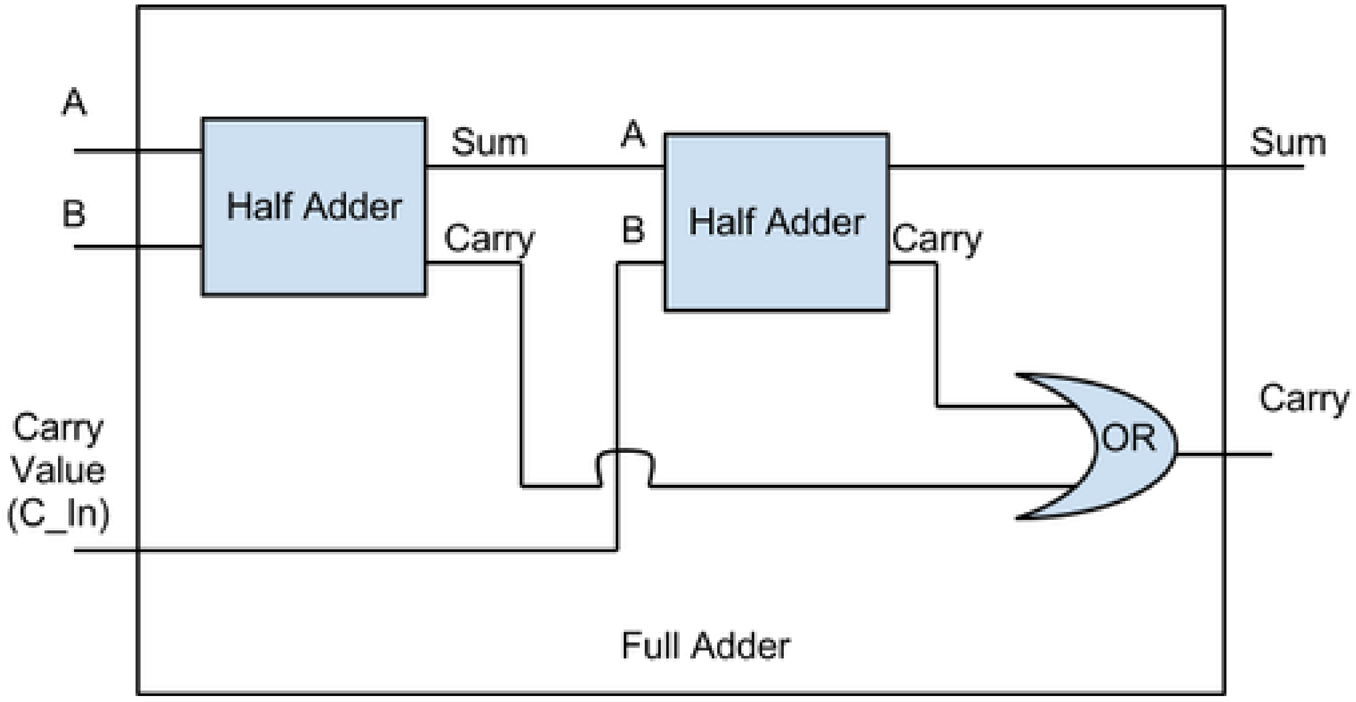


Figure 30 - Full Adder

This Full Adder incorporates the Half Adder, the basic logic circuit that handles the addition of two binary inputs, and turns it into a complete adder. I won't go into detail with the Full Adders, but make sure you understand where all the inputs and outputs are going.

We can combine Full Adders together to create what's called a Ripple Carry Adder, which will be able to take binary input and accurately output the resulting addition. Here is an example of a bunch of Full Adders coming together to make a Ripple Carry Adder.

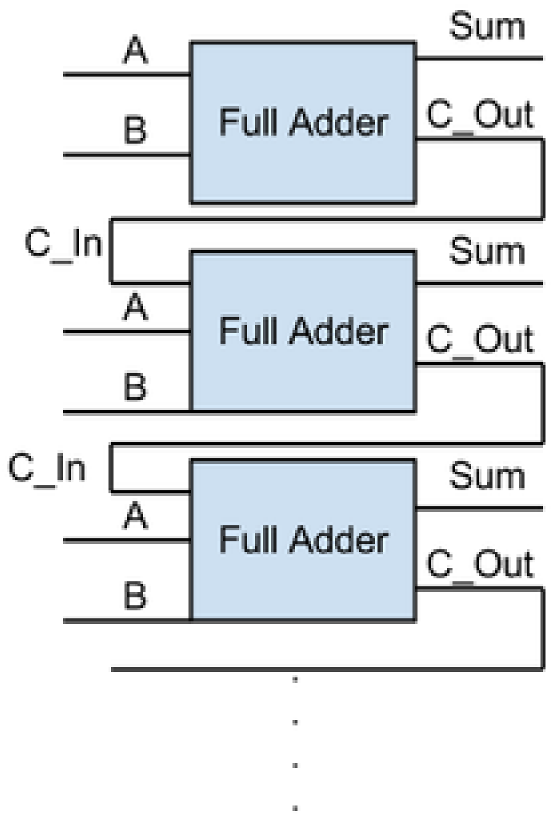


Figure 31 - Ripple Carry Adder