Study Guide

Basics

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.

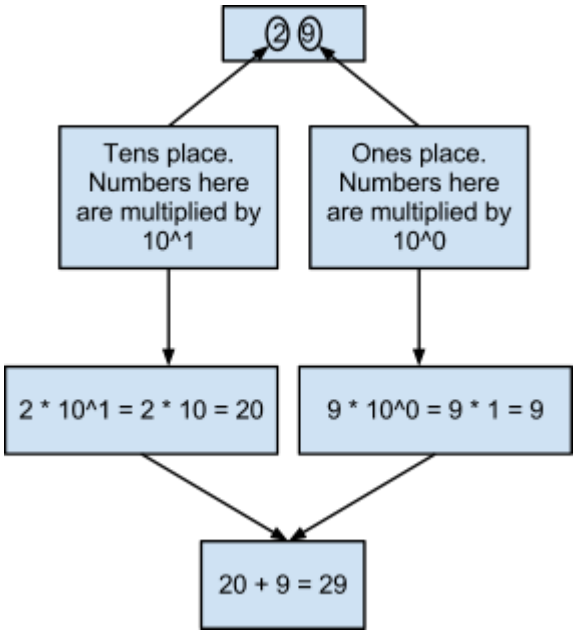


Figure 1 - Base-10 representation of 29

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.

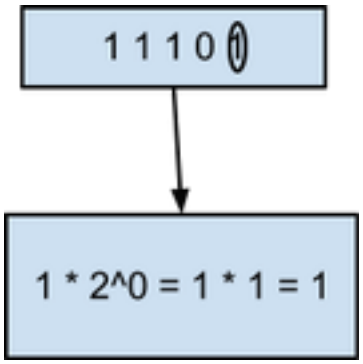


Figure 3 - Binary ones place

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

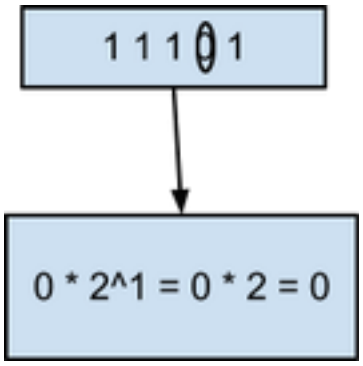


Figure 4 - Binary tens place

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

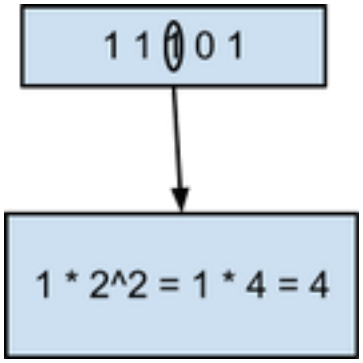


Figure 5 - Binary hundreds place

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.

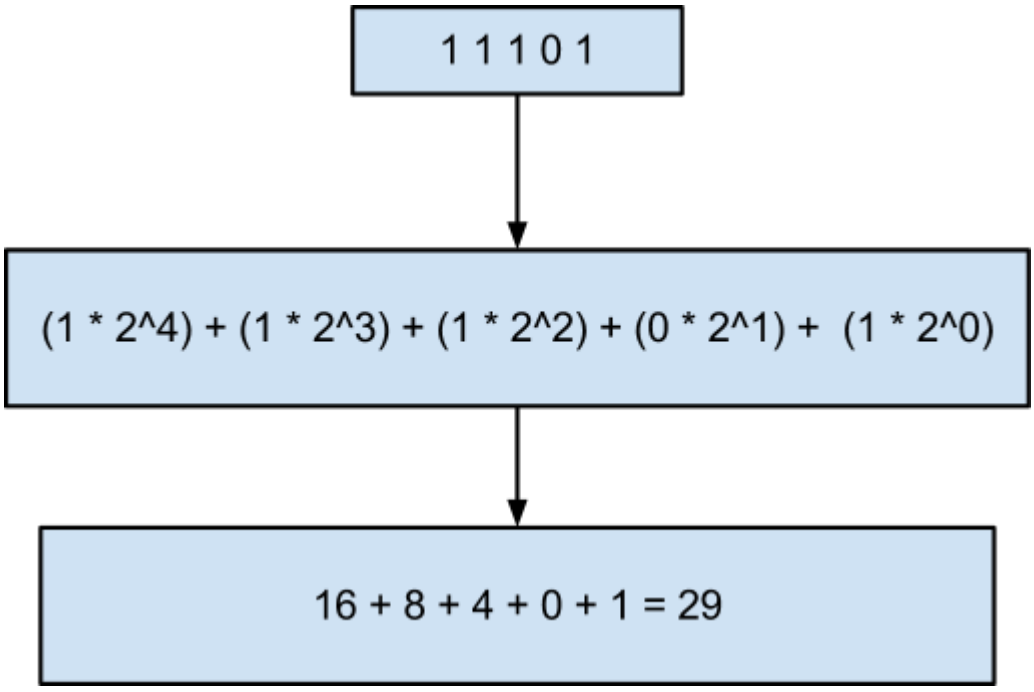


Figure 6 - 29 in binary

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.

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.

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:

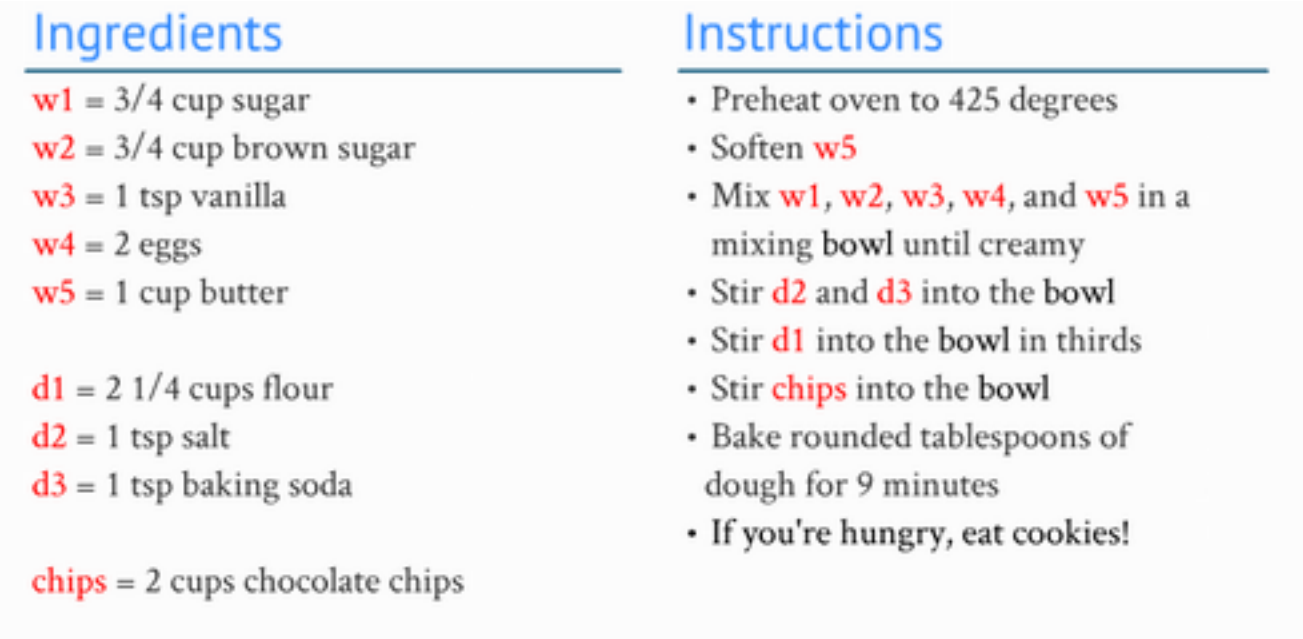


Figure 7 - A simple list of cooking instructions[1]

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:

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

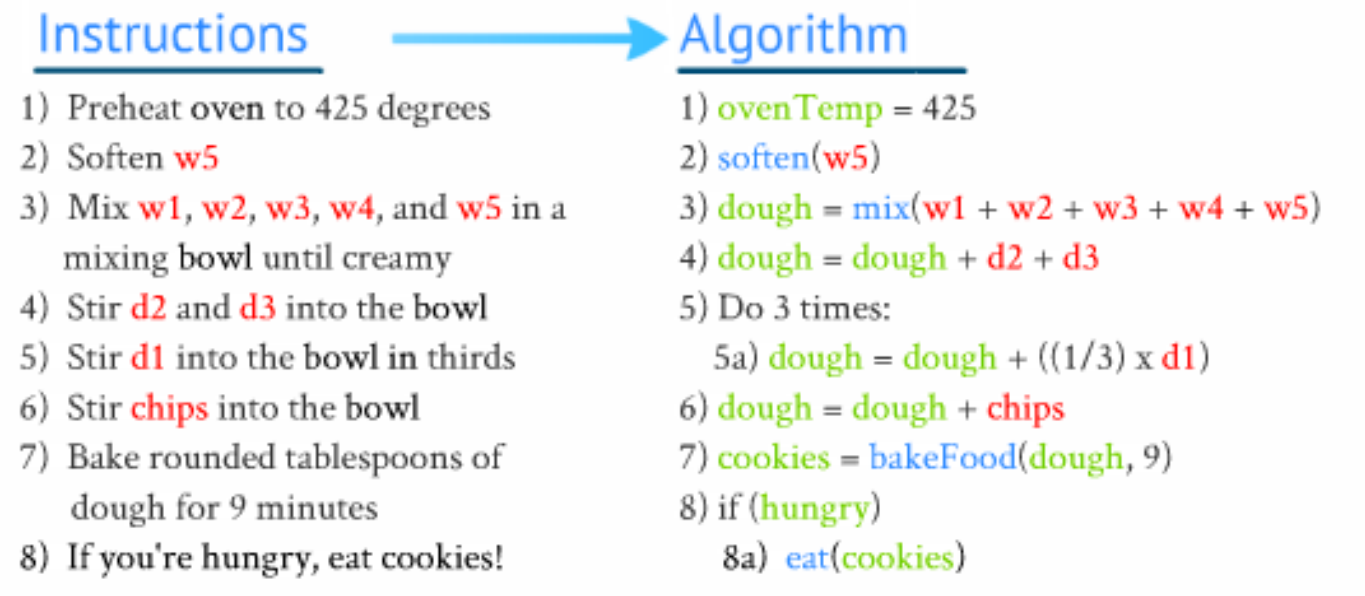


Figure 8 - Cooking instructions turned into an algorithm

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

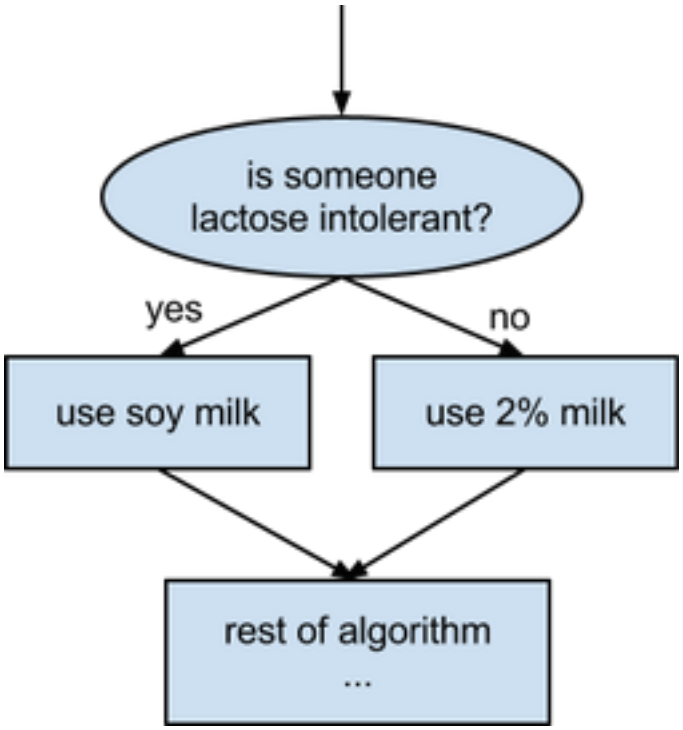


Figure 9 - selection statement

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)
4) dough = dough + d2 + d3
```

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).

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:

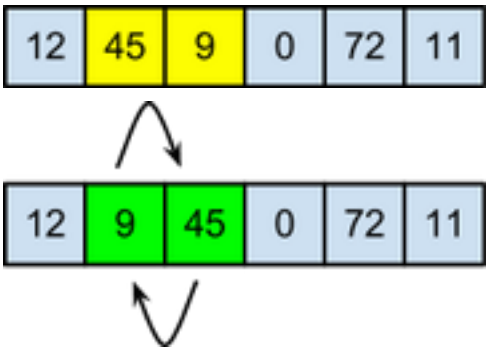
We pass through the array the first time, swapping two items when the item on the right is a smaller number than the object on the left. We will iterate through the entire array once.

Step 1

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

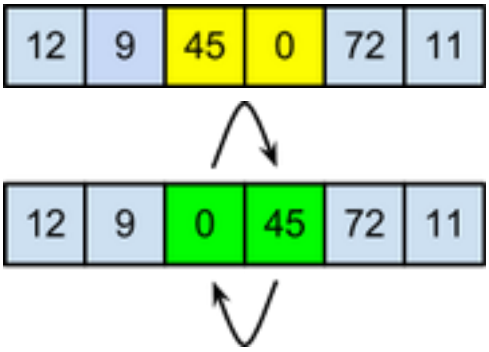
Step 1.1

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



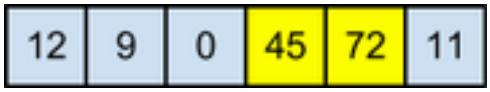
Step 1.2

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



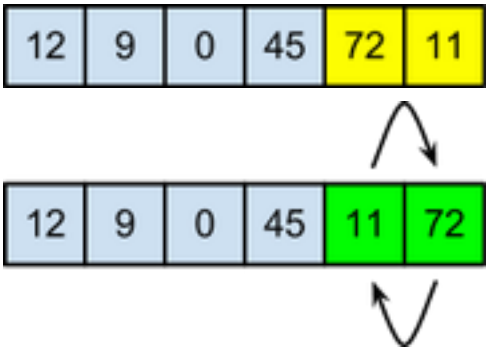
Step 1.3

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



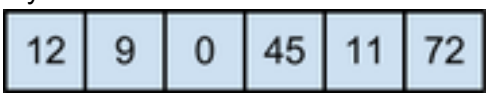
Step 1.4

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:



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.

End of step 2

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 pseudo code 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 pseudo code[2]

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.

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 bubble sort is

$$O(n)$$

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

Data Storage

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

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 11 - 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 11 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 12 - 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 12, 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.