Bits, Bytes, Building With Binary

Vaidehi Joshi · Follow
Published in basecs · 12 min read · Jan 2, 2017

If you work with computers (or even if you don't!), there's a good chance that you've heard people talk about computers as just "a bunch of ones and zeros". This was one of the few things I knew about computers before I got into software: *it's all just ones and zeros*. It was only after I learned to code and started programming professionally that I realized what that really meant. Yes, computers run on ones and zeros. It's definitely a bit more complicated than that, but it's not so complicated that we can't understand it!

Let's start by giving our problem a name. Those ones and zeros that computers are made up of? Those are based on a type of number system called **binary**. The binary number system hinges on a simple idea that, instead of counting with 10 digits — the way that we learned to do in kindergarten — you can count with just two digits. The binary number system that is used in computers today was created by Gottfried Wilhelm Leibniz in 1679, but this way of counting has a much longer history that dates back to the ancient Egyptians.

Okay, so, if binary has just two digits, how do you count past…two?

Binary counting

https://giphy.com

In our modern day counting system, we have ten possible digits per place. This is why we sometimes hear people refer to our counting system as *base 10*; another name for it is *denary*. In the binary number system, we have *two* possible digits per place, so we can refer to it as counting in *base 2* (which sometimes abbreviated as *bin* for binary). The number of digits possible per place is the only real difference in the way that we count in base 2 versus base 10.

Base 10

digits: 0 – 9

counting: increment digits in one place until you can't anymore, then add a digit to the next place

example:

20	10	0
21	11	1
22	12	2
23	13	3
...	14	4
	15	5
	16	6
	17	7
	18	8
	19	9

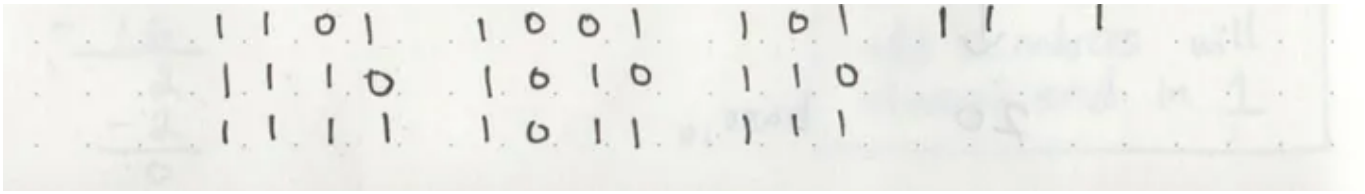
Base 2

digits: 0, 1

counting: increment digits in one place until you can't anymore, then add a digit to the next place

example:

1100 1000 100 10 0



Counting in base 10 versus base 2

In the example to the left, we can see that we start off counting the same way we do in base 10. First with 0, then with 1. But when we get to the number 2, how do you keep counting?!

Well, let's think about what we do in base 10. When we get to the number 9, what do we do? We reset the units to start again with the number 0, and increment our tens place to the number 1. When we've gone through all the possibilities between 10–19, we reset the units place to 0, and increment the tens place to the number 2. We do this until we've reached the numbers between 90–99, and then we add another place: the hundreds place.

The same logic applies to counting in binary. Start with 0 and 1. To represent the number 2, we reset the first place to 0 and add another digit to the left: 10. Then we increment the first place again: 11. And if we continue to do that, we'll see that the first 10 numbers in binary like this:

0
1
10
11
100
101
110
111
1000

1001
1010
1011
1100
1101
1110
1111

(Pssst — there's a pattern in the number of possible permutations/combinations per digit! But if you don't see it yet, don't worry. It should become a little bit more obvious later).

Writing in binary

We know that computers run on binary. And yet none of us type binary into the keyboard! This would lead us to believe that, somehow, what we type into our machines gets converted (compiled) down to binary. This happens through *several* layers of abstraction, and we won't get into all of them.

While it's not important to know all the layers, I do think that there's value in knowing a little bit about how that conversion works. We'll keep it simple and focus on converting between base 10 numbers (integers) to binary.

Remember in elementary school when we all had to learn our multiplication tables? And then remember in middle school when we started learning exponents and realized how useful those multiplication tables were? Well, get ready to re-realize that once again! I've been practicing converting to and from binary a lot over the past week and I've realized that the most important thing you can do when it comes to learning binary is brush up on your powers of two. (But just in case you need a little help, I've included the powers of two in my examples below.)

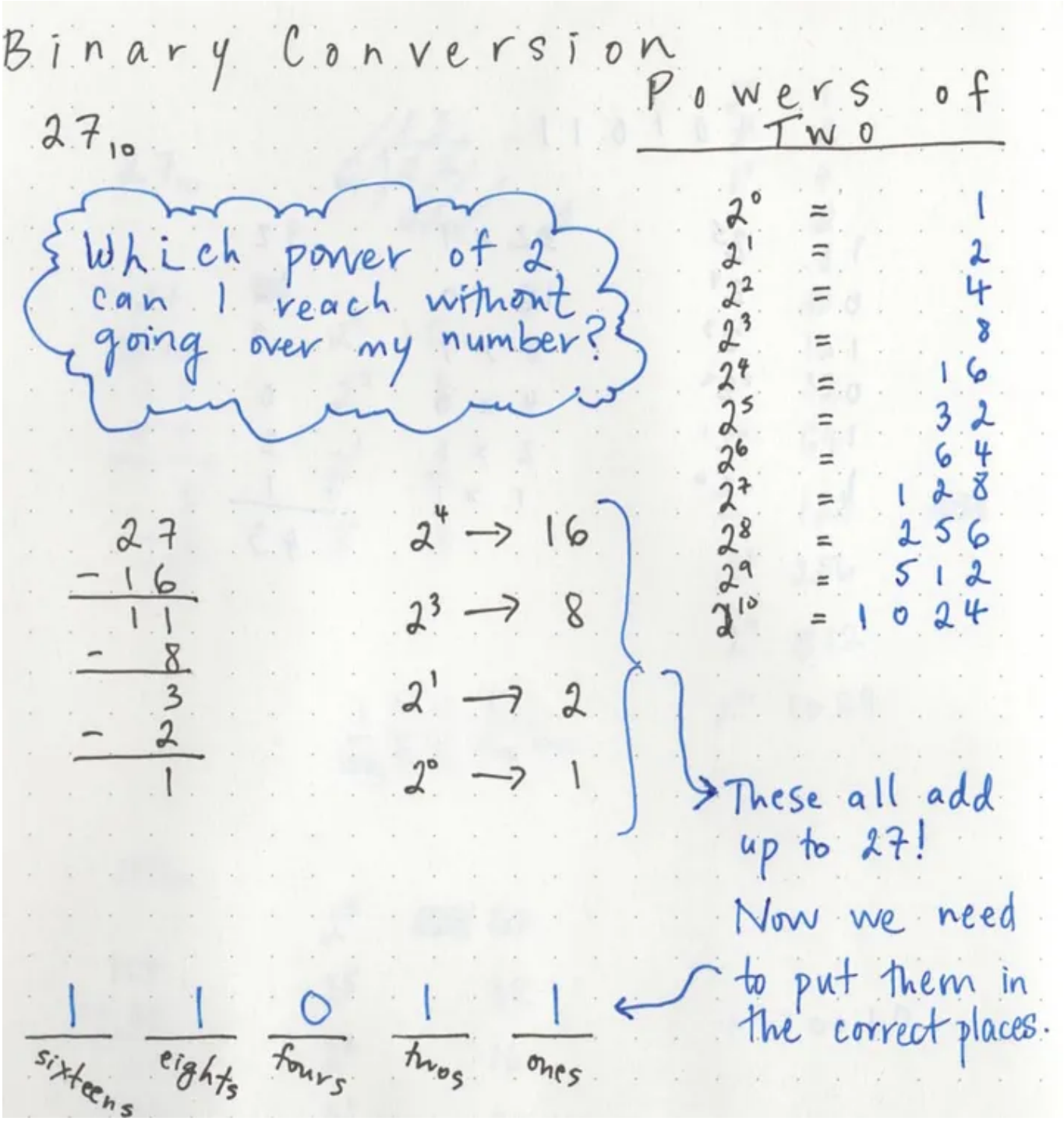
Converting into binary

Let's take a look at a couple examples. First, let's try converting the number 27 (in base 10) into binary (base 2).

What we want to do is break this number down into powers of two. So we can ask ourselves: *which power of 2 can I reach without going over the number that I want to convert?*

Once we find that number, we want to subtract it from our total amount, and then repeat this process until we are left with zero. An important thing to remember (which I always seem to forget): 2 raised to the power of 0 is *always equal to one*!

This might make more sense to see in an example:



Converting 27 into binary

Once we've broken down our number into powers of 2, we need to put them in the correct place. In base 10, we have *units*, *tens*, *hundreds*, *thousands*, and so on. In binary, we our places come from — you guessed it — the powers of two. Our places will be: ones, twos, fours, eights, sixteens; in other words, 2 to the power of 0, 2 to the power of 1, 2 to the power of 2, and so on.

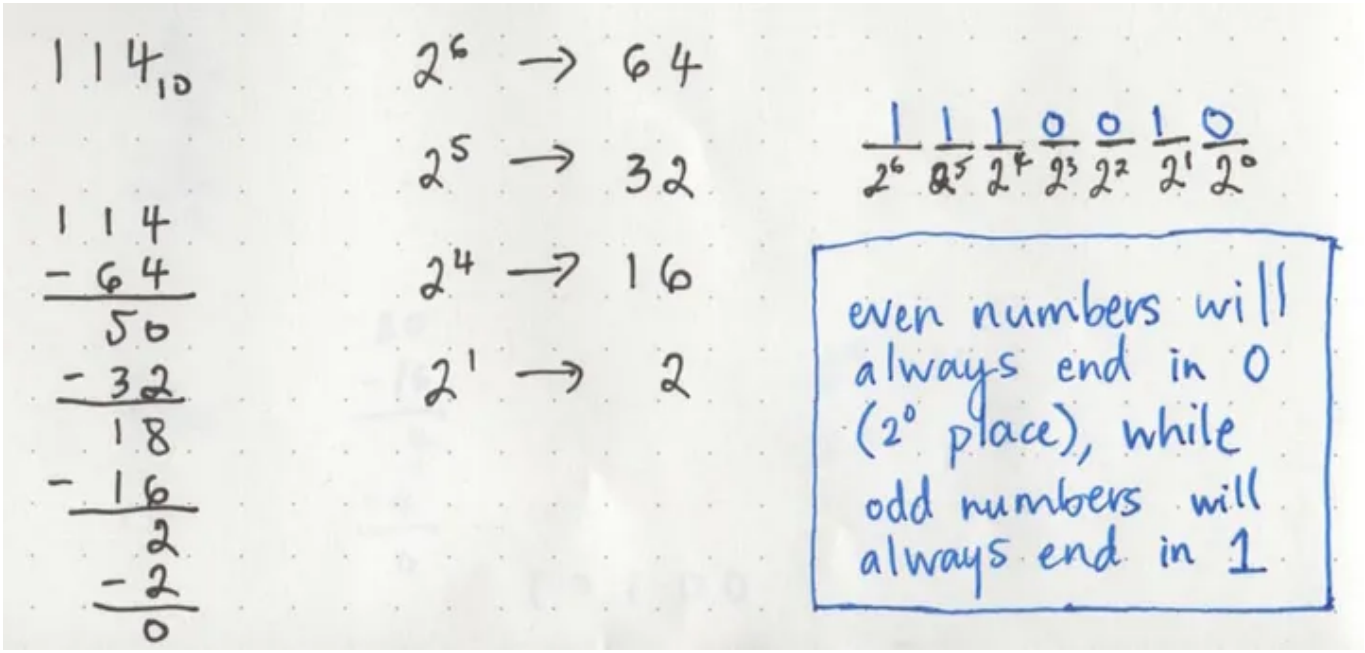
We can see that we have a value of 16, a value of 8, a value of 2, and a value of 1. That's exactly how we know which places these numbers belong in! We'll want to put a 1 in each of these places, and any place/power of two that *doesn't* have a value in our number breakdown will get a zero. Since nothing in our number 27 could be broken down into a power of 2 (that is to say, we didn't have any 4's in our number breakdown), we'll put a zero in that place.

And that's it! The number 27 can be converted from base 10 into binary as: **11011.**

Okay, one more example. Let's go big this time — the number 114.

The highest power of two that we can get to without going *over* the number 114 is 64, or 2 to the power of 6 (2 to the power of 7 is 128, and that's too big because it goes over our number!). We immediately know that we're going to have a 1 in the place for "64", or "2 to the power of 6".

Let's keep breaking the number 114 down into powers of two:



Converting 114 into binary

Okay, so we ended up with a 64, a 32, a 16, and a 2. Once we put a 1 in the appropriate place settings, we end up with this number:

1110010

And that's it! The binary equivalent of 114!

If we do enough binary conversions, we'll start to notice that even numbers in base ten will always end in 0 when they're converted into binary. Conversely, odd numbers in base ten will always end in 1 when written in binary. Remember that rule I mentioned earlier? ***2 raised to the power of 0 is always equal to one*!**

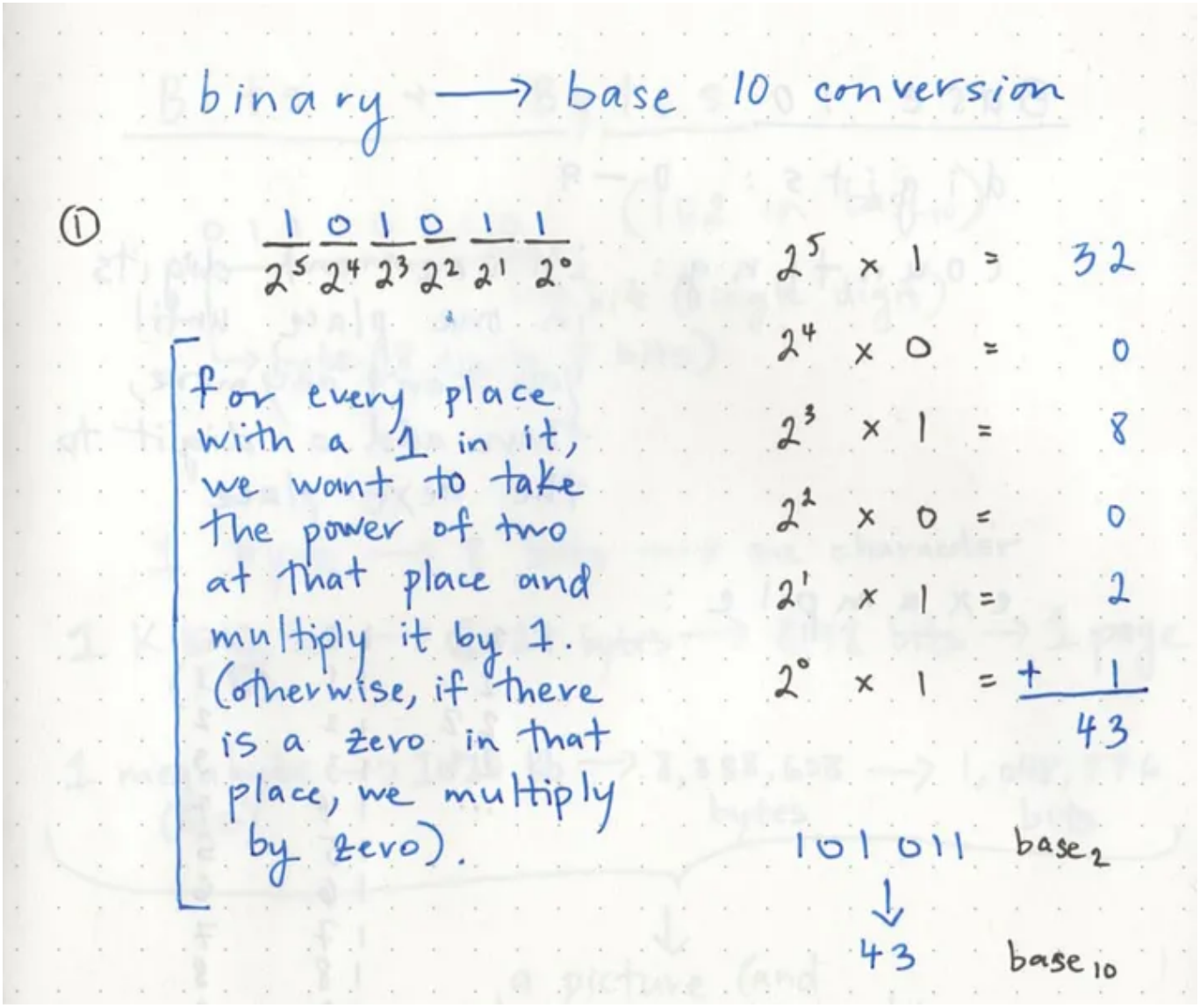
That's that rule coming into play. Even numbers are divisible by two, which means that you'll never have an extra remainder of 1 when you're breaking down your number into powers of two.

Converting out of binary

Interpreting numbers from binary is a lot easier once you know how to write in it. When we were converting from base 10 into binary, we were breaking things down into powers of two, right? But what we were *really* doing was ***dividing by powers of two***.

Based on that logic, we'll do the exact *opposite* thing if we want to convert binary *into* base 10. That is to say: we'll *multiply by powers of two*.

Let's take a look at what that might look like; we'll convert **101011** into base 10:



Converting 101011 into base 10

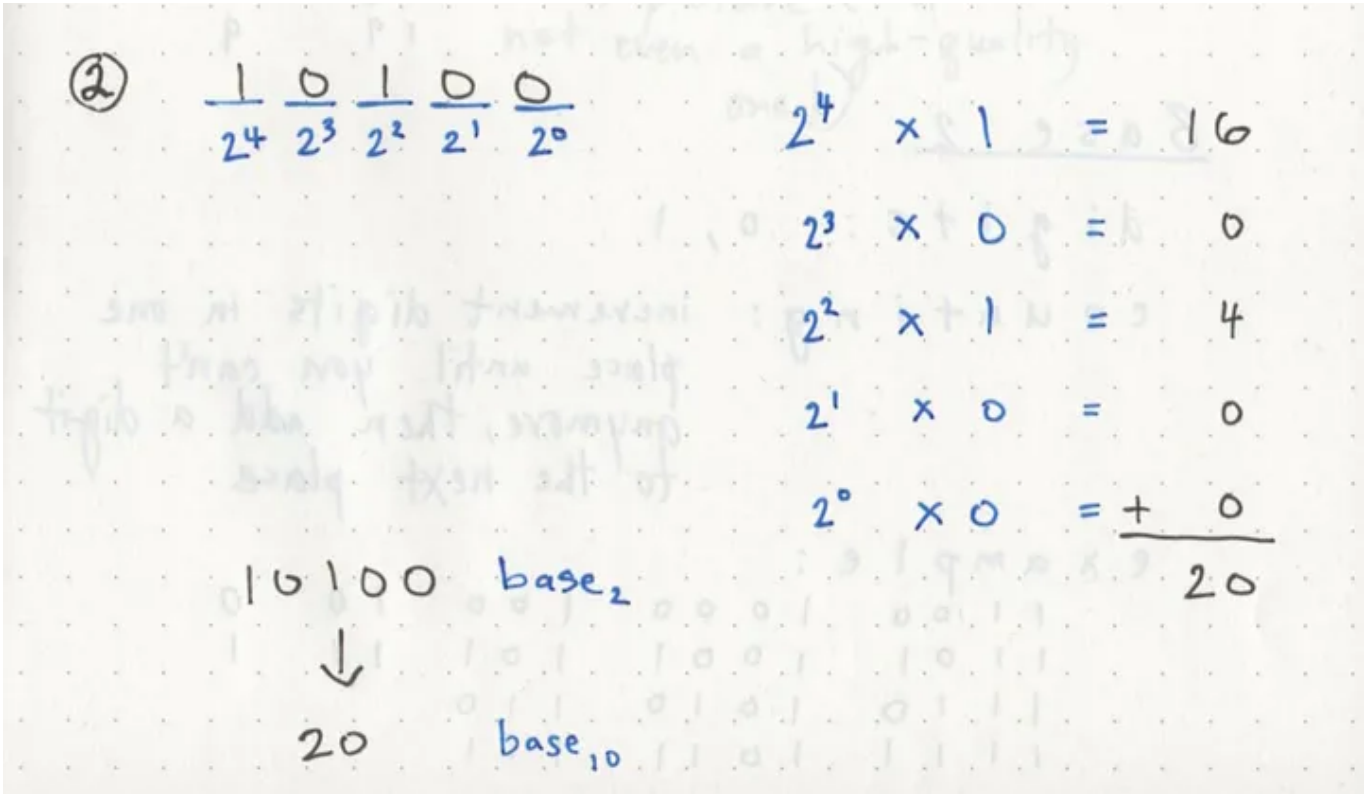
First, we'll look at what's in each place and remember which power of 2 that place is associated with. We can start from left to right: we see that we have a 1 in the 32's place (2 to the power of 5), which means we know that the base 10 version of this number can be broken down (read: divided) into 32.

We'll write that down on the side, and keep going. There's a zero in the 2 to the power of 4's place (which we could also call the 16's place), so instead of multiplying is by 1, we'll multiply it by 0.

A good rule of thumb when converting out of binary is this: *if there's a one in the place, multiply the power of two for that place by 1 (and then keep doing that until you've gone through all the places!).*

Eventually, we get down to the last digit and end up these numbers: 32, 8, 2, and 1 (which is the same as 2 to the power of 5, 2 to the power of 3, 2 to the power of 1, 2 to the power of 0). All of those numbers combined gives us the base 10 conversion of 101011: the number **43.**

One more quick example — this time, let's try a smaller number. Here's how we'd convert **10100** into base 10:



Converting 10100 into base 10

This one is a little bit easier to understand, hopefully! We might even be able to do it in our head (depending on how well we know our powers of 2). We

know that we're going to want to sum the value of 2 to the power of 4 and 2 to the power of 2.

Well, by now we probably can do this pretty quickly: 2 to the power of 4 is 16, while 2 to the power of 2 is 4. What's 16 + 4? 20.

And there you have it! 10100 is the same as **20** in base 10. Easy peasy!

How do computers read binary?

Okay, enough with the math. What does this have to do with computers?

At the heart of it, computers are made up of switches. We already know that computers interpret binary. But what we might not realize is that the switches and circuits that are the building blocks of computers today are effectively representations of binary.

A computer has billions of (super tiny) digital circuits, which are incredibly simple. They are made up of switches. And a switch can only have two states: **on** or **off**. Another way to think about this is **true** or **false**. And we can represent that on/off binary in *yet another* way: 1 and 0.

> Binary is the numbering system that computers use in order to represent on and off. On is 1, and off is 0.

What's even cooler is that everything in computers (and computer science, at that!) is, at the most rudimentary level, based on this on/off paradigm. Little bursts of electricity either pass through or don't pass through based on whether something is switched on or switched off.

https://giphy.com

So how does a computer interpret and break down complex things (like this blog post, for example) into just ones and zeros? It uses different units of measurement, which all can be converted into binary.

A single digit in binary is known as a ***bi**nary digit*. But, you might know it as a **bit.** Since we know that binary is base 2, and one digit can only ever be either a 0 or a 1, we also can deduce that a **bit** can only ever be comprised of either a *0 or a 1*.

What that means is that our computer has to do everything by building binary numbers, which means using only 0 and 1 and stringing them together. Which seems kind of insane! But it can build bits on top of other bits. And that's exactly what it does. It strings together 8 bits (8 digits) into a **byte**. We might have already heard the term "byte" thrown around, or perhaps seen it on Stack Overflow. A **byte** is so common in the way that computers interpret binary that it is considered a *unit of computer memory*.

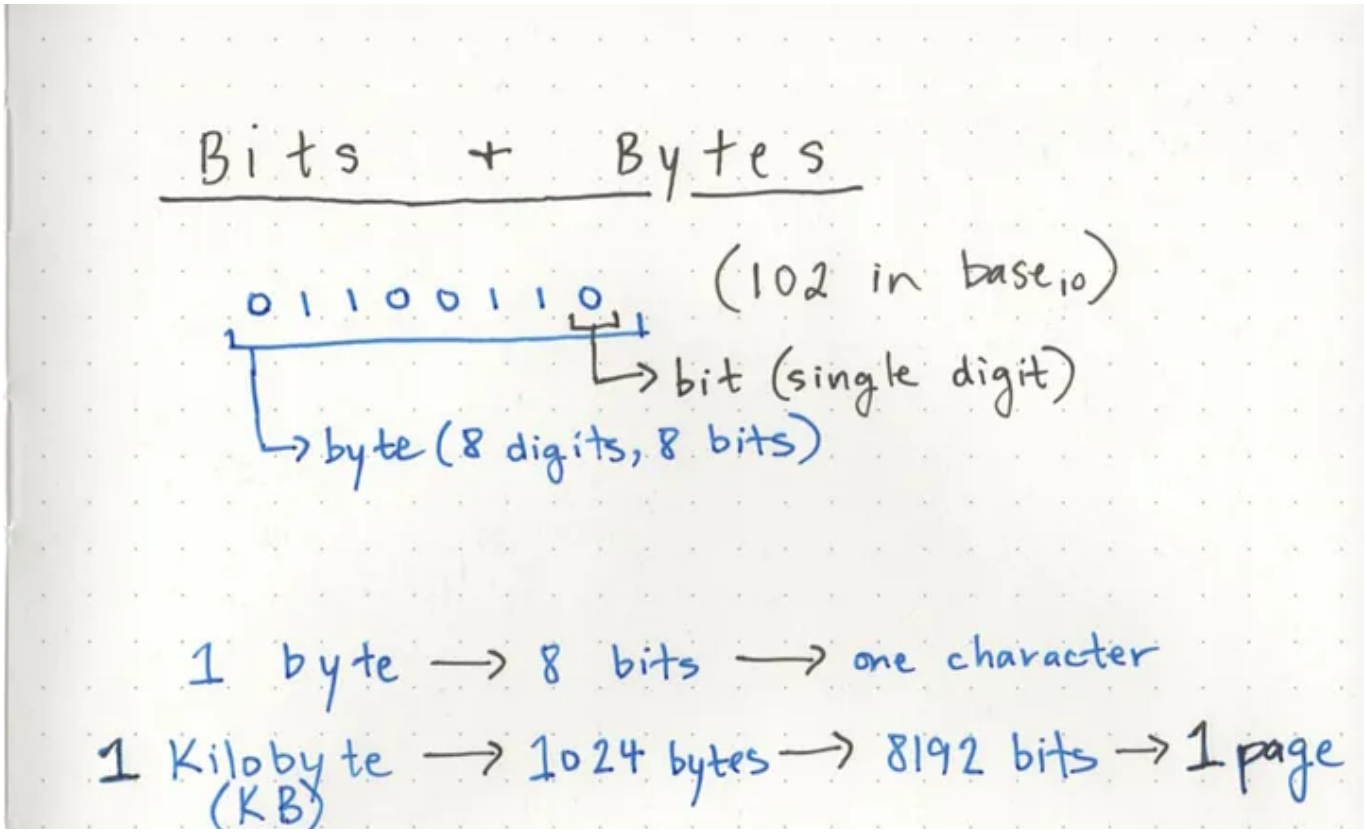
I think that bytes are particularly interesting because a single byte can represent 256 different combinations. (*Remember powers of 2? 2 to the power of*

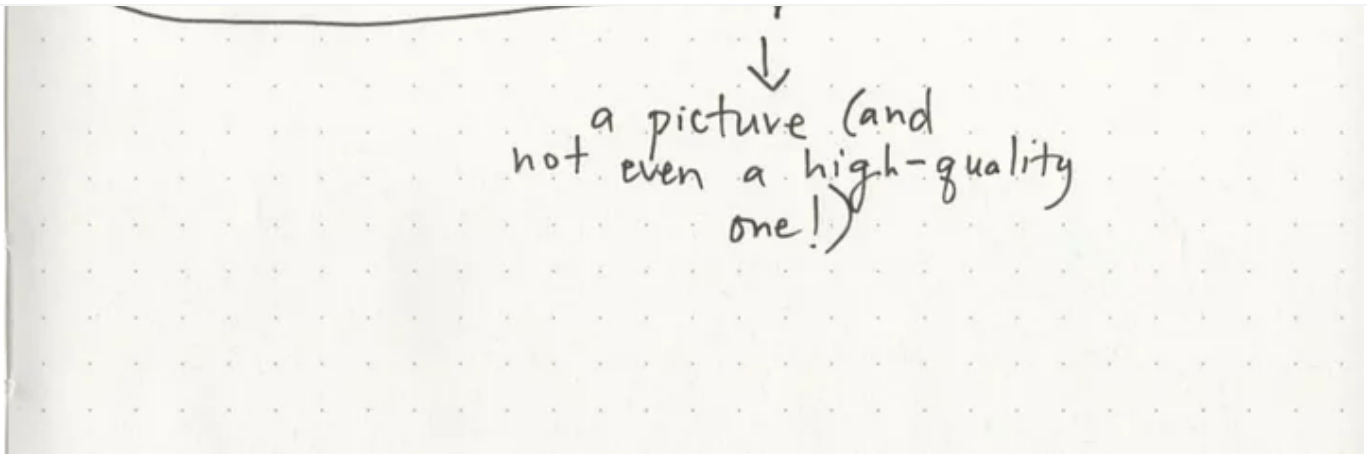
8 is 256.) And what about if you have two bytes? Two bytes means 16 bits (binary digits), which means that you can represent 65,536 different combinations (2 to the power of 16)! That's a whole lot of different permutations that you can represent with just 2 bytes! If you think about a single circuit (often called *transistors*) handling an on/off switch per digit, just 16 transistors can process and interpret a ton of information!

Bits, the building blocks of bytes, are incredibly fundamental and worth understanding. They're important because different computers can process a different number of bits at a time. An 8-bit machine, for example, breaks up and processes 8 bits at a time. A 16-bit machine would break up and process 16 bits at a time. The number of bits that are processed at a time are known as a **computer word**, so we can think of *bits* as the "letters" that make up a computer word. Most computers now have a word length of 32 or 64 bits. And now you know what that means: that your machine passes around and processes 32 or 64 bits at a time. In other words, your computer processes binary strings that are 32 or 64 digits long!

These units of computer memory are what people are usually referring to when they say "everything is just ones and zeros". Because, truly, it is.

Let's take a single character of a word. That character requires 8 bits (or 1 byte) in order to represent it. So, what about something longer? What about a page of text that's somewhere around 1,000 words long? That would require a lot more bytes!





CORRECTION: 1 megabyte is actually 1,048,576 BYTES and 8,388, 608 BITS!

Maybe I've been staring at too many ones and zeros, but it seems like once you start thinking about the scale of bits and the ways that they're strung together and constructed and used...well, everything starts looking like binary!

The power of two

Binary is something that few programmers think about these days. Deep down, we know that it's important and worth learning about, yet can seem so overwhelming and kind of unnecessary to think about.

But if you think back to when computers used to take up entire rooms (imagine how big the circuits and transistors were back then!) and how far they've progressed and how much they've changed since then, it's rather jaw-dropping.

At the very core of that, is binary — language that every computer speaks and understands. So if you're interested in or work with computers, the basics of binary is worth knowing a little bit about. After all, even though it's just two numbers, it is, ultimately, what the world around us is written in.

http://www.someecards.com

Apologies in advance if you start dreaming in 0's and 1's now!

Resources

If you found this post interesting, check out these resources below. I found these very helpful in learning about binary, so perhaps you will, too! Happy learning.

1. Intro to Programming Course on binary, Boston University
2. Mathematics lessons on binary conversion, Khan Academy
3. Powers of Two, Vaughn Aubuchon
4. The story of 256, Gray Watson

Programming | Computer Science | Tech | Software Development | Fundamentals

Written by Vaidehi Joshi

Follow

29K Followers · Editor for basecs

Writing words, writing code. Sometimes doing both at once.

More from Vaidehi Joshi and basecs