KEVIN FOCKE

THE CODE monster MANUAL

Vol 6

"I want to order exactly 6 breads," said the Exactor.

"Quack, did you say 9 breads? How generous of you to support a hard-working breadwinner..."

THE CODE MONSTER MANUAL VOLUME 0

NUMBERS, TYPES, STORAGE, AND ABSTRACTION.

COMPUTER FUNDAMENTALS FOR AMBITIOUS BEGINNERS.

THE BOOK I WISH I HAD.

In the Code Monster Manual we learn the concepts & pitfalls that every programmer should know, independent of programming languages and WITHOUT needing to be a math wizard.

Surrealistic drawings and cover background by Darío Mekler.

Monster drawings and cover cartoon by Thais Freire Sanchez.

Uses the Fira Code font and Fluent Emoji.

This is a Wildly Creative Book by Kevin Focke.
Find more projects at kevinfocke.com

TABLE OF CONTENTS

CHAPTER 0: Intuition

NO NEED TO BE A MATH WIZARD

How can you prevent the 125 million dollar mistake that caused the NASA Climate Orbitor to crash?

You don’t need to be a rocket scientist or mathematician to understand it. In fact, the mistake was very simple: They misunderstood what a number was.

Anyone can make this mistake and we will learn how to prevent it.

Now you might think: “I still have a math trauma from high school. Do we *really* need to learn about numbers?”

Fret not, dear astronaut. You don’t need to be a math wizard or multi-dimensional alien to be a great programmer. This book does not hide insights under cryptic mathematical equations. Nor is there even a single line of code in the whole book. Instead, we start from first principles as we build up our knowledge of timeless computing concepts that are relevant in whichever programming language(s) you end up using.

“Why do I need to learn about numbers, anyway? I just want to make programs. Can’t we just use functions for that?”

Biologists learn about the core principles of atoms, even though they are more interested in the structures emerging from atoms: Proteins, cells, etc.

Similary, a function, and more generally software, emerges from numbers. Yet, unlike the inflexible rules of chemistry, the rules of programming are not limited by functions; we gain the flexibility to change how the functions themselves work by deeply understanding numbers!

“But coding bootcamps promise me I can learn how to code in 30 days or less. And they don’t bother me with numbers, either.”

The complexity of computers is hidden away by programming languages and operating systems. This gives a false sense of confidence until, one day, you encounter problems that can only be solved by deeply understanding how the computer itself works—including numbers.

Sure, you can make programs until you encounter those problems, but that’s like paddling your surfboard into a whirpool… Fun, if you’re a fan of suddenly sinking to the bottom of the terrible abyss. Invitations to “The

Sinking Crew” are available in an “easy” programming book or bootcamp near you.

Furthermore, the book is spiced up with eclectic insights because concepts are often relevant beyond the boundaries of an industry or field.

For example:

`Here Be Monsters`, the old naval maps used to say about uncharted territories. It’s helpful to know where the dangers are, but software is not a stormy sea nor a dark alley in a city. Software has no location; it’s not a physical building you can walk in. Software is more like air; it’s everywhere.

Computers are monsters because they think different than us. The Code Monster Manual helps you understand where the monsters lurk, and gives practical guidance on preventing monster mayhem. The book provides a helicopter-view of the most bang-for-your-buck coding concepts.

While I try to keep the content approachable for ambitious beginners, there is quite a lot of ground to cover and the pace is fast. It’s natural to have a lot of questions in the beginning and it takes time to fully answer them.

Look, you may even feel discouraged when learning about computers because the subject is so broad. But remember then: Nothing worthwhile is ever easy. A mountain is climbed step by step, and so it is with computers.

Along the way, you’ll learn about bright ideas and bits of trickery.

Bright Idea

Repeat what you’ve learned in your own words to better retain information.

Bit Of Trickery

Incremental change is hard to notice, and yet it’s the basis of most progress. Perfection is for the gods and the delusional. Us mere mortals, we iterate!

As software developers do, we start counting the volumes from 0. Why do developers count from 0? It’s related to a positional starting point called an `index`. The position you are currently at is 0. If you move forward one step from the starting point, you are at position 1. If you move backwards one step from the starting point, you are at position -1. Later, we will learn how to represent negative numbers with positive numbers which sounds paradoxical—and yet that *is* how we do it.

In the 0’th chapter, we will first build our intuition with some dogs, before graduating to misunderstood monsters.

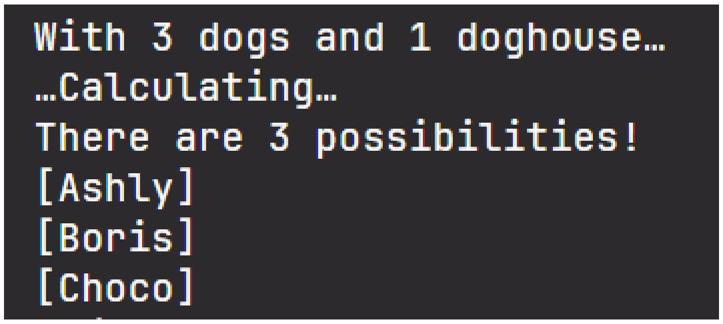
LET SLEEPING DOGS LIE

Somewhere in the Savanna, there are 3 dogs; Ashly, Boris, and a puppy called Choco. The caretaker Johhny starts an animal shelter for the dogs. He builds a lovely little doghouse and paints it with blue clouds. He then builds another doghouse, painted in a gnarly green. Boris grumps a little.

Johhny starts to wonder, as programmers do: How could I assign these dogs to their doghouses?

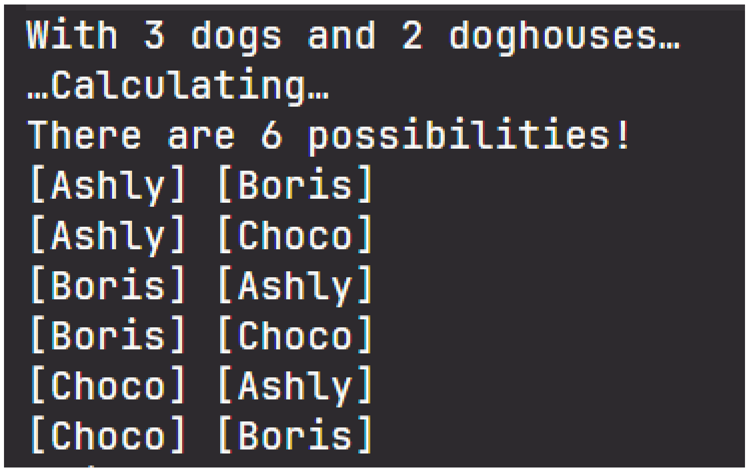
“So… let’s think about it… A dog can either be in a doghouse, or roam in the Savanna. If a doghouse is available, it *must be assigned to a dog*. Unfortunately, I can’t put multiple dogs in the same doghouse because they’re very territorial animals.”

Johhny writes up a program to calculate the possibilities and it spits out the answer:



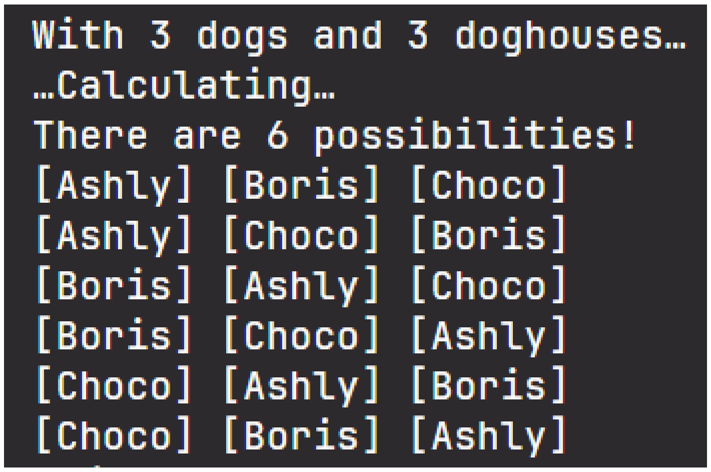
```
With 3 dogs and 1 doghouse…
…Calculating…
There are 3 possibilities!
[Ashly]
[Boris]
[Choco]
```

Well that was obvious. But what happens if we add more doghouses?



```
With 3 dogs and 2 doghouses…
…Calculating…
There are 6 possibilities!
[Ashly] [Boris]
[Ashly] [Choco]
[Boris] [Ashly]
[Boris] [Choco]
[Choco] [Ashly]
[Choco] [Boris]
```

Johhny builds a third doghouse, painted with red roses. Boris *woofs*.



```
With 3 dogs and 3 doghouses…
…Calculating…
There are 6 possibilities!
[Ashly] [Boris] [Choco]
[Ashly] [Choco] [Boris]
[Boris] [Ashly] [Choco]
[Boris] [Choco] [Ashly]
[Choco] [Ashly] [Boris]
[Choco] [Boris] [Ashly]
```

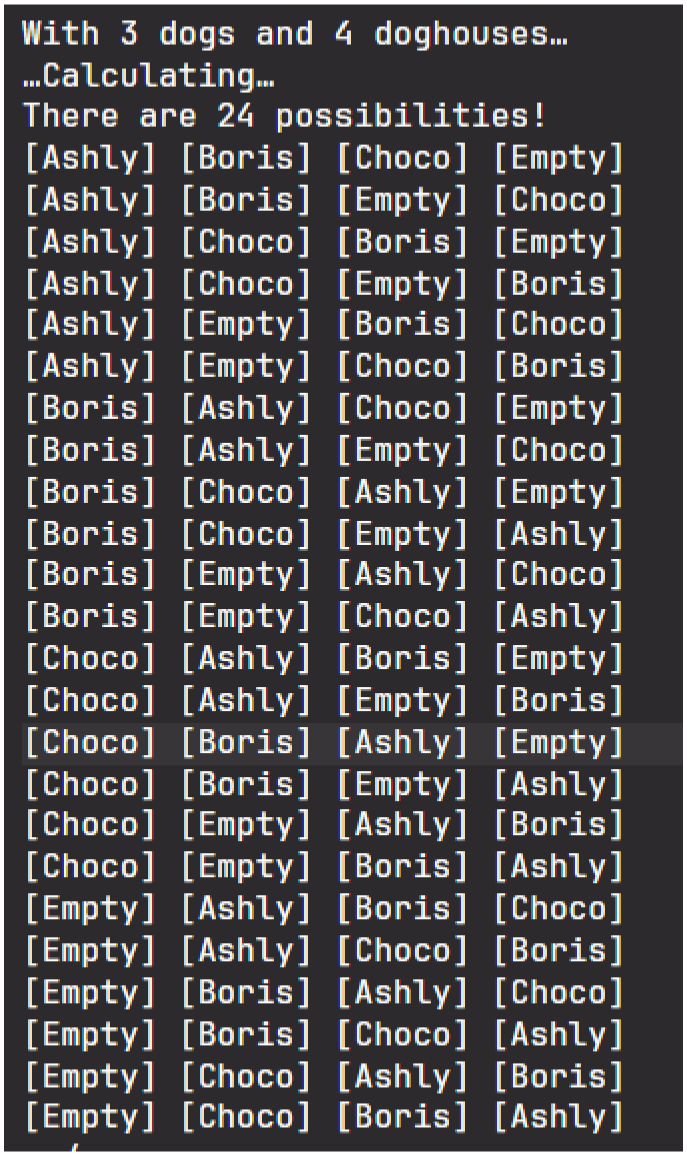
You may notice there are still 6 possibilities—how can that be? What is missing?

Well… we have pretended dogs don't exist unless they're inside a doghouse. Later on, we will see that the concept of `absence` is so tricky for programmers that it cost "a billion dollars of pain and damage…"

Johhny builds another doghouse and paints it purple. Boris starts barking loudly. He is mad and trying to communicate, although he finds it a tad difficult because he's a dog. If dogs could talk, Boris would say:

"Before I was happy with any doghouse; it gets cold here at night. However, now there are more than enough doghouses and I want the one painted with roses! And if I can't have that one, I'd rather have the one with the blue clouds than the gnarly green.”

Johhny runs the program.



```
With 3 dogs and 4 doghouses…
…Calculating…
There are 24 possibilities!
[Ashly] [Boris] [Choco] [Empty]
[Ashly] [Boris] [Empty] [Choco]
[Ashly] [Choco] [Boris] [Empty]
[Ashly] [Choco] [Empty] [Boris]
[Ashly] [Empty] [Boris] [Choco]
[Ashly] [Empty] [Choco] [Boris]
[Boris] [Ashly] [Choco] [Empty]
[Boris] [Ashly] [Empty] [Choco]
[Boris] [Choco] [Ashly] [Empty]
[Boris] [Choco] [Empty] [Ashly]
[Boris] [Empty] [Ashly] [Choco]
[Boris] [Empty] [Choco] [Ashly]
[Choco] [Ashly] [Boris] [Empty]
[Choco] [Ashly] [Empty] [Boris]
[Choco] [Boris] [Ashly] [Empty]
[Choco] [Boris] [Empty] [Ashly]
[Choco] [Empty] [Ashly] [Boris]
[Choco] [Empty] [Boris] [Ashly]
[Empty] [Ashly] [Boris] [Choco]
[Empty] [Ashly] [Choco] [Boris]
[Empty] [Boris] [Ashly] [Choco]
[Empty] [Boris] [Choco] [Ashly]
[Empty] [Choco] [Ashly] [Boris]
[Empty] [Choco] [Boris] [Ashly]
```

Johhny assigns the dogs to their house, gives them food adapted to their diets, and goes to sleep.

In the middle of the night, all hell breaks loose! Boris scares away Ashly and takes over her house. Boris celebrates his victory by eating Ashly’s meal and promptly has an allergic reaction.

Bright Idea

Differentiate between a mild allergic reaction `a recoverable error` and a horribly allergic reaction `an unrecoverable error`. The unrecoverable error is also called a `panic`. This distinction matters a great deal to programmers and the dogs they love.

Ashly complains to her friend Choco, interrupting Choco's beauty sleep—and now Choco is mad too! Ashley completely panics and runs away into the Savanna.

Johhny wakes up—such chaos! He falsely believed the situation was stable, but the dogs moved around. So much trouble with just 3 dogs and 4 doghouses. Now imagine adding more dogs and doghouses. Yeah, that'd be a mess…

In a nutshell: Everything in a computer is a bunch of numbers. These numbers can represent different things like dogs, the food they eat, or even doghouses.

It gets complicated because the same number can represent different things, different numbers can represent the same things, and the numbers can move around causing all sorts of chaos.

If this makes no sense yet, don’t worry; we have a whole book left to explain!

A bonus round, in case you’re curious…

Question: how many possibilities are there with 3 dogs and 10 doghouses?

Answer: 720 possibilities!

Q: 3 dogs and 20 doghouses?

A: 6 840 possibilities!

Q: 3 dogs and 30 doghouses?

A: 24 360 possibilities!

Q: 4 dogs and 30 doghouses?

A: 657 720 possibilities!

Q: 5 dogs and 30 doghouses?

A: 17 100 720 possibilities!

Q: 6 dogs and 30 doghouses?

A: Way too many possibilities, it crashes the computer!

There might be a way to improve Johhny’s program so it doesn’t crash, but that’s not the point. The point to remember is that the amount of possibilities, or `state space`, keeps increasing at an ever-faster pace. This will become relevant in a later volume. For now, let’s focus on understanding numbers.

HIGH-LEVEL PROGRAMMING VS. LOW-LEVEL PROGRAMMING

“Developers, Developers, Developers,” Steve Ballmer, then-CEO of Microsoft, famously chanted at the turn of the millenium. Yet all software developers must ultimately bow down to: Numbers.

It bears repeating: EVERYTHING inside a computer is controlled by numbers. We have a whole book to learn about *many* nuances. Here’s a little teaser:

A number can be a count of things (5 rabbits), it can represent an ordering (the 3rd duck), it can represent a location (lives in rabbithole number 42), or a time (Bunion, The Rabbit, woke up at 3 AM in the morning). Even a function itself is composed of a bunch of numbers that carry out actions by activating hardware circuits with instruction numbers.

Numbers, numbers, numbers.

To avoid thinking about numbers, programmers abstract away the underlying complexity of numbers by creating simpler interfaces. Similar to how a subway map shows the routes while hiding the jaggedness of the real routes. To keep an overview & navigate better, we hide information that is not relevant.

There is a distinction between `high-level programming` and `low-level programming`.

High-level programming uses more abstractions whereas low-level programming uses fewer abstractions. In other words, low-level programming works closer to the foundation. Both kinds of programming are useful in different scenarios.

High-level programming is useful when the pre-defined abstractions align with the functionality you want; why reinvent the wheel? In contrast, low-level programming is the most difficult and useful when you want maximum flexibility.

Low-level programming requires you to understand how the system *actually* works rather than understanding simplified abstractions built on top of it. High-level programming is like learning the buttons of an all-in-one cooking robot—handy if that's the functionality you need. The book series is thus focused on low-level programming.

SUMMONING THE NUMBEROMICON

In volume 0, *The Numberomicon*, we are laying lifelong software foundations by thoroughly understanding what a number is. Similar to constructing a building, laying the foundation of drab gray concrete is not the most interesting part. And yet, it is essential if you want to build something impressive.

You have summoned the Numberomicon.

Chapter 1, The Exactor: Why do computer need exactness and why do they use binary numbers?

Chapter 2, The Contraducktion: The versatility of numbers is confusing; the same number can represent different things, and a different number can represent the same thing. How can we avoid this confusion and stop thinking about numbers?

Chapter 3: Where and how are numbers stored?

Who can access them?

Chapter 4: Computers are ultimately a tool. The underlying complexity is abstracted away to improve usability; the messiness is hidden and you can always peel back another layer. What are the broad considerations of using abstractions at different scales?

CHAPTER 1: THE EXACTOR

Spirit Of The Monster: Computers do exactly what you tell them to. If you expect the computer to just act reasonable, you will be surprised by the results…

FABLE: AN AMBIGUOUS ENCOUNTER

“Okay. What *exactly* do you mean with that?” the Exactor asked for the umpteenth time.

“Well look, I just want you to add these numbers,” said the Programmer.

“Okay. What do you mean with `just`?” the Exactor pondered.

“Ignore that word,” said the programmer, curtly.

“Okay. What do you mean with `add`?”

“Add means you combine one or more numbers.”

“Okay. What do you mean with `numbers`?”

“A number is… well it’s obvious isn’t it?” Responded the Programmer, nervously tapping his foot as if he’s a drummer in a long-haired metal band.

“Unfortunately, it’s not obvious at all, sir. Could you explain what a `numbers` is?”

“A number is how we count an amount of something,” said the Programmer.

“Okay. And what do you mean with `numbers`?”

Asked the Exactor.

“I just explained…”

“With all due respect, I don’t think that you did sir nor did you answer my question.” The Exactor said with a slight air of vicarious embarrassment.

“What?!” The Programmer shouted in disbelief.

“Please keep your tone down,” said the Exactor, calmly. “You explained what a `number` is and I am very appreciative that you did. But it still remains unclear what a `numbers` is. A `numbers` is not a `number`, is it?”

“You’re making a joke aren’t you?”

“No, sir. You see, I want to avoid *any* possibility of ambiguity. Please explain: What is a `numbers` *exactly*?”

WHY ARE COMPUTERS EXACT?

A hungry donkey was stuck between two big piles of hay. Both piles looked delicious, and the donkey did not know what to choose! So, being the silly donkey that he is, he kept standing in the middle.

On the internet, nobody knows if you're a donkey, but even we humans are familiar with the difficulty of making decisions.

For example, in a supermarket, we struggle choosing a snack. Is it time for popcorn? Perhaps some crisps? Or, something savoury like goat cheese? Or maybe… maybe some chocolate. But then… white chocolate, milk chocolate, or pure chocolate? And with or without nuts? Pecans or peanuts?

There is a parallel in neuroscience. It is impossible for a perfectly "rational" agent to choose the best option among two equally appealing options. Luckily, we have emotions & preferences to nudge us in some direction.

Indecision is okay when it concerns snacks, but some decisions must be made quickly—in a snap. When you’re driving a car and the light turns yellow, do you speed through, or hit the brakes? You need to know NOW.

Similarly, programming is the art of telling a computer how to make quick, consistent decisions. You must tell the computer all relevant information and give *exact*, step-by-step instructions about how to deal with a situation; the fancy name for a step-by-step solution to a problem is an `algorithm`.

However, the real world is messy and selecting the right information is a trainable skill.

For example, counting the amount of clouds is not very useful for predicting rainfall; where does one cloud begin and another end?

Instead of counting clouds, the atmospheric pressure is measured, the saturation of air with liquid, how often it rains this time of year, among other factors…

The art of science is choosing which information you take into consideration. Ideally, you use the information that actually matters—that *caused* the phenomenon—and ignore the rest. In other words, you extract the significant `causation` variables and conduct an experiment in a controlled environment.

Yet the outside world forever refuses to be defined; it’s like a child continually asking “why?”. There is always another layer of explanation or another nuance.

“Why this?”

“Because that.”

“And why is that?”

“Because of another reason.”

“And why is that?

When your favourite little gremlin keeps asking these annoying questions, you eventually give up and respond: “Because *that* is how it is.”

At some point, you must simply accept things as they are, even though a conceptual circumstance is always somewhat incomplete and/or inconsistent.

Thus, if you want to make quick decisions with a computer, you must be able to exactly explain the steps with an `axiomatic` (assumed-to-be-correct) concept represented by, you guessed it: A number. More specifically, a discrete number.

The hardware of your computer translates the messiness of the real world into something you can think about. It does this by converting a voltage to a specific number.

For example:

- 0.25 volts would be converted into the number 0.

- 0.75 volts would be converted into the number 1.

- What happens to a voltage of 0.5? That’s a problem you luckily don’t need to think about.

In other words, software developers don’t need to worry about the `impedance mismatch` between the continuous voltage number, and the discrete number that it gets converted to; in software we are *always* using a discrete number.

As a kid, I hated hearing the sound of an ambulance because I knew something bad happened. Now, I’m happy there’s an ambulance coming to help.

Much in life is about perspective and

computers are no different.

Bit Of Trickery

How can you divide a cake into three equal parts? There are two approaches:

- The Continuous approach; each slice of a cake is 1/3rd of that cake, or 0.33333333333… cakes; it is a continuous number that goes on forever and strikes fear into the hearts of even the most mathematically gifted wizards because we’re facing the messiness of the real world. The ancient Greeks called them `irrational numbers`, and it is said that Hippasus, the person who discovered them, was drowned at sea for his sins. Computer nerds also call it a `Floating Number` which might be a reference to Hippasus… floating somewhere.

- The Discrete approach; we can view a cake as a collection of cake slices. So a cake that we’re cutting into three pieces is 1 cake with 3 slices. Taking 1/3rd of that cake means taking 1 whole slice. This is called an `integer`; in Latin, “integer” means something which is whole and undivided.

In other words, we can choose how we view a problem, and choosing the right lens can be the difference between a never-ending headache and a problem solved easy-as-pie.

For simplicity, throughout the book, we assume a discrete number is used. In practice, that’s mostly—but not completely—true.

Here’s a droll joke: What’s the difference between theory and practice?

All theory fails in practice, given enough time and scenarios. We constrain the world to something we can understand—but no abstraction is ever complete. Indeed, abstractions *require* that we ignore something—that’s what makes them useful.

Anyhow, what *exactly* is a discrete number?

WHAT EXACTLY IS A NUMBER?

Let’s start this chapter with a little challenge; write your definition: What is a number?

Seriously, take the time to write it down. It does not need to be perfect; the worse it is, the more you will learn!

Ready?

“A number is how we count an amount of something.”

Incomplete, a number can be used for more than counting:

- It can represent a phone number: “Call 555-2368 for problems with your plumbing.”

- It can represent your position in a queue; an ordering: “You are number 5. Please wait before buying your bagel.”

- It can represent a ranking: “Daxtogenics is the nr. 1 nose spray with fresh mint smell.”

- It can represent a temperature. “It’s a hot day in sunny Palm Springs, 77 degrees Fahrenheit. That’s 298,15 Kelvin for our European viewers.”

- It can represent a time. “It’s 10 before 12.” AM or PM?

- It can represent an item on an eating menu: “Number two, please.” “Okay, how spicy should the squid be, madam?” “Not terribly.”

“A number is a series of 0’s, 1’s, 2’s, 3’s, 4’s, 5’s, 6’s, 7’s, 8’s, and 9’s.”

Not quite. A number can also be written in different ways; the number `10` can be written as `ten`. Furthermore, most of the world does not speak English and writes out full numbers differently. For example, in Spanish the number `10` is `diez`.

Bright Idea

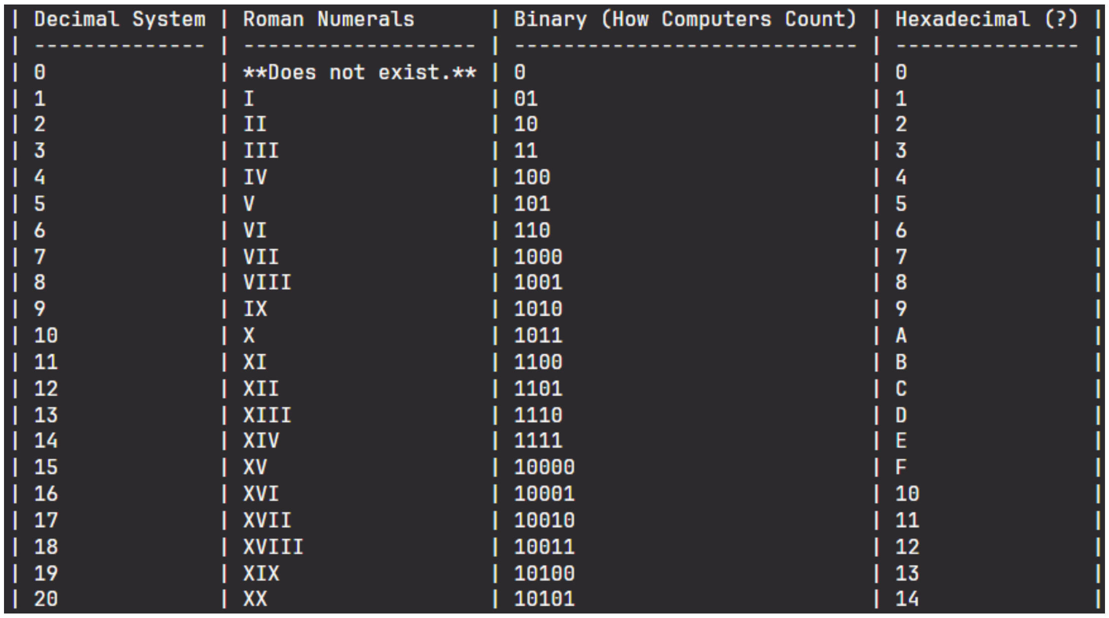
Be careful with accents in Spanish. Año means year *with* the accent. And *without*? Well… that is somewhat related to a donkey…

Moreover, there are different numbering systems (as we will further explore in the next chapter.) People tend to be most familiar with the digital system. Literally, you count it on your hands. A “digit” is a synonym for finger.

When you’ve counted to 10 in the digital system, you start again with empty hands. Similarly, modern numbering systems move one position to the left once they reach their `base`. A numbering system with 10 fingers is called `base-10`. If you lose a hand, you become a `base-5` numbering system. If you only have 2 fingers in total, you are a `base-2` or `binary` numbering system.

A 16-fingered mutant, would be a `base-16` or `hexadecimal` numbering system.

Let’s count from ‘zero’ to ‘twenty’ in different numbering systems:



Decimal System	Roman Numerals	Binary (How Computers Count)	Hexadecimal (?)
0	**Does not exist.**	0	0
1	I	01	1
2	II	10	2
3	III	11	3
4	IV	100	4
5	V	101	5
6	VI	110	6
7	VII	1000	7
8	VIII	1001	8
9	IX	1010	9
10	X	1011	A
11	XI	1100	B
12	XII	1101	C
13	XIII	1110	D
14	XIV	1111	E
15	XV	10000	F
16	XVI	10001	10
17	XVII	10010	11
18	XVIII	10011	12
19	XIX	10100	13
20	XX	10101	14

In other words, 10 can either mean `ten` or `two` depending on which numbering system you use. And `20` does not exist in binary! But what is binary number, *exactly*?

BINARY NUMBERS

The smallest piece of information you can hold in a system is called a `bit`. In the case of computers, that’s a 0 or 1. You can imagine observing a tiny light bulb; it is either `on` or `off`.

The Exactor does not accept dimmable lamps because they are too ambiguous.

Thus, there are *exactly* two possible scenarios:

```
`0` Light 1 is `off`.
`1` Light 1 is `on`.
```

Programmers call these scenarios a `state`. Computers change states with a bunch of `transistors` (part of the processor). Why are they called transistors? Because they are a component that transitions between states! Maybe that’s obvious to you, but nobody ever explained that to me!

Bit Of Trickery

The phenomenon of losing the origin of a concept over time is called “semantic diffusion.” For example: Did you know that films were silent for a long time? When sound was finally introduced, people talked in films so they called them `talkies`. How silly! Luckily we don’t use the term `talkies` anymore. But wait. Hold on. What’s another name for a film? A movie! It’s something that moves! And what’s a film? Well it’s named after the plastic film that movies were recorded on! And why is a computer called a computer? Because it computes—it calculates something! And why is an operating system called an operating system? Well it’s related to telephones; the word `telephone` is composed of the Greek words `tele` and `phone`; `tele` means “at a distance” and `phone` means “sound”. The telephone creates sound at a distance. And how were the telephones connected? They were connected by people who operate (set up) the system. These employees were called operators. And why is the operating system Windows called Windows? Because the key concept of Windows is that you put things in windows. And why is Apple called Apple? No idea, they must be geniuses.

There are 2 (bi) possible states per transistor; we thus call it a `binary` system. If there were 10 (deci) possible states per transistor, we could call it a `decimal` (or `decinary`) system.

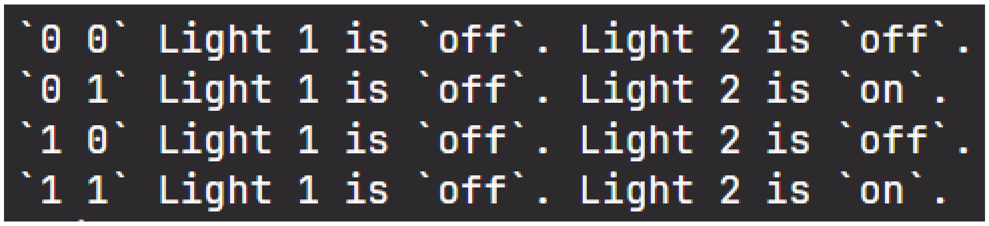
In other words, we use a binary numbering system because that’s how much state information can be held by a transistor.

Bit Of Trickery

While extremely uncommon, computers based on 3 states do exist; they are called `tritary` computers.

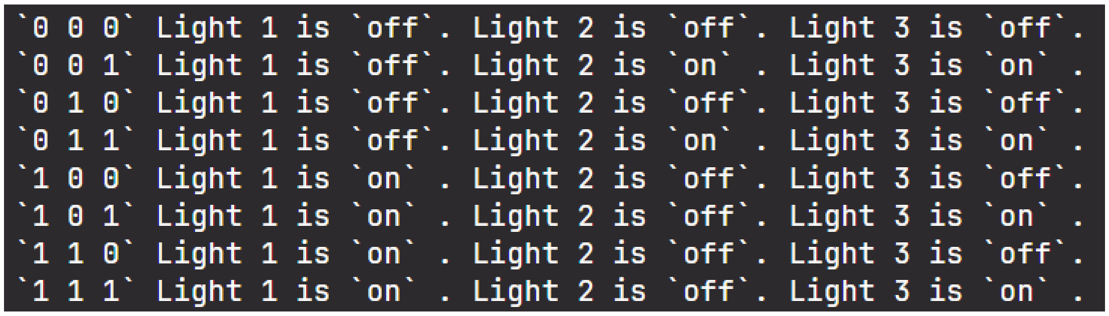
Now for our next trick; we add more lightbulbs.

With 2 lightbulbs there are 4 possible states:



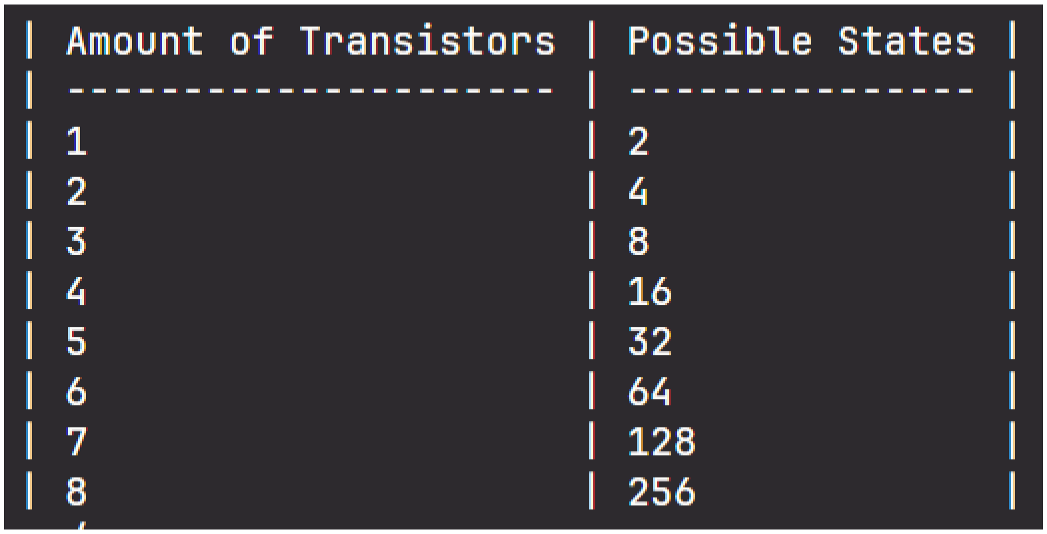
```
`0 0` Light 1 is `off`. Light 2 is `off`.
`0 1` Light 1 is `off`. Light 2 is `on`.
`1 0` Light 1 is `off`. Light 2 is `off`.
`1 1` Light 1 is `off`. Light 2 is `on`.
```

With 3 lightbulbs there are 8 possible states:



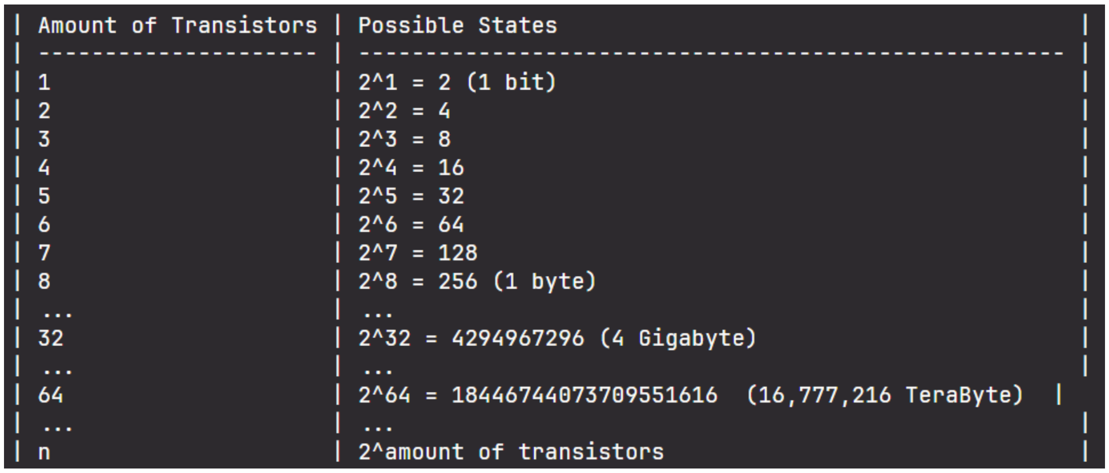
```
`0 0 0` Light 1 is `off`. Light 2 is `off`. Light 3 is `off`.
`0 0 1` Light 1 is `off`. Light 2 is `on` . Light 3 is `on` .
`0 1 0` Light 1 is `off`. Light 2 is `off`. Light 3 is `off`.
`0 1 1` Light 1 is `off`. Light 2 is `on` . Light 3 is `on` .
`1 0 0` Light 1 is `on` . Light 2 is `off`. Light 3 is `off`.
`1 0 1` Light 1 is `on` . Light 2 is `off`. Light 3 is `on` .
`1 1 0` Light 1 is `on` . Light 2 is `off`. Light 3 is `off`.
`1 1 1` Light 1 is `on` . Light 2 is `off`. Light 3 is `on` .
```

Here is a table showing the amount of transistors and the possible states. Can you see the pattern?”



Amount of Transistors	Possible States
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256

What about now?



Amount of Transistors	Possible States
1	2^1 = 2 (1 bit)
2	2^2 = 4
3	2^3 = 8
4	2^4 = 16
5	2^5 = 32
6	2^6 = 64
7	2^7 = 128
8	2^8 = 256 (1 byte)
...	...
32	2^32 = 4294967296 (4 Gigabyte)
...	...
64	2^64 = 18446744073709551616 (16,777,216 TeraByte)
...	...
n	2^amount of transistors

By noticing the pattern, we can reduce the amount of things we need to remember. That’s why math can be great! You can get things done quicker. Unfortunately, a lot of math teachers ignore the steps that lead to the insight and just give you a formula.

“Why did you add 32 & 64 transistors?”

A teaser for a future unknown. :)

EVERYTHING IS A NUMBER

Truly, it’s no overstatement to say that EVERYTHING stored inside software is composed of binary numbers that mean something in a particular context. But just to drive the point home…

When you move your mouse, a number represents the `vertical position` on your display and another number represents the `horizontal position.`

The little graphic image of your mouse cursor is also a bunch of numbers, representing the pixels.

What happens if you tap a key? That’s another number!

What happens if you left click on your mouse? That’s another number!

What happens if you right click on your mouse? That’s another number!

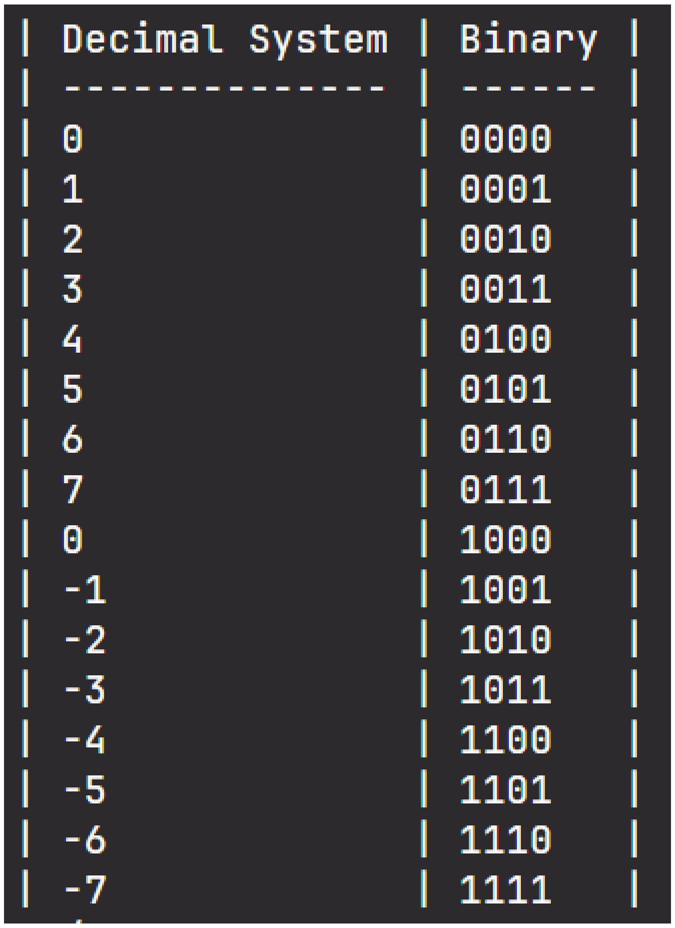
What happens if you put a finger on a touch screen? That’s a bunch of numbers!

Take a guess, what happens if you change the volume slider?

That’s a bunch of numbers! Each little movement on the slider is yet another number, as is each movement of your mouse.

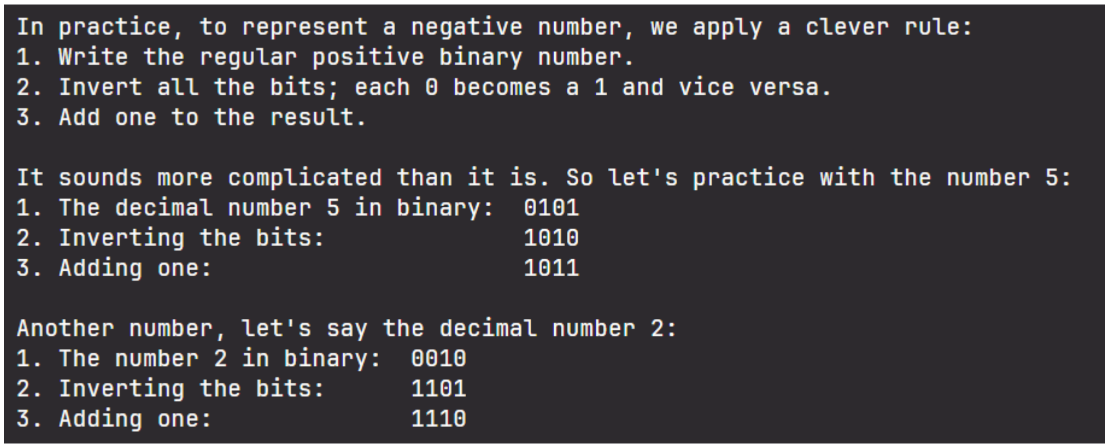
“What about negative numbers?”

We can use a binary number to represent the sign; 0 for a positive number and 1 for a negative number. Here is how that works:



Decimal System	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

Unfortunately, this representation of negative numbers stores the decimal number 0 twice. This is both wasteful and error-prone. For example, how could we compare the binary numbers `0000` and `1000`? Those are not the same binary number, even though they represent the same decimal number `0`.

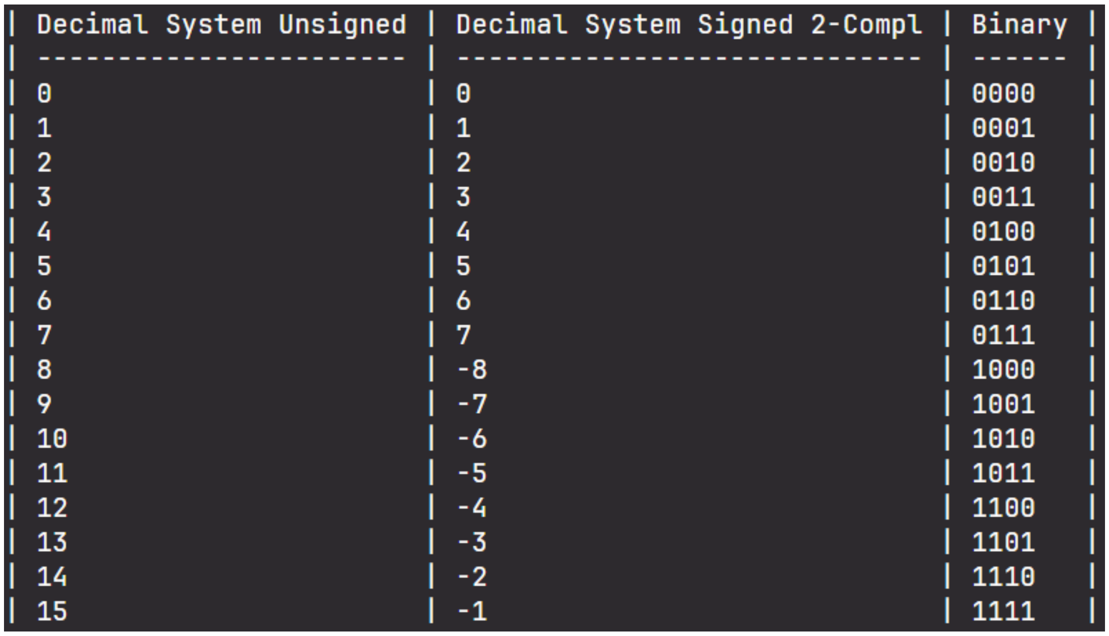


```
In practice, to represent a negative number, we apply a clever rule:
1. Write the regular positive binary number.
2. Invert all the bits; each 0 becomes a 1 and vice versa.
3. Add one to the result.

It sounds more complicated than it is. So let's practice with the number 5:
1. The decimal number 5 in binary:  0101
2. Inverting the bits:              1010
3. Adding one:                      1011

Another number, let's say the decimal number 2:
1. The number 2 in binary:  0010
2. Inverting the bits:      1101
3. Adding one:              1110
```

In the end, we have an extra number: -8. Efficiency! This is called the `2-complement` representation of a signed integer. Right they are! Whoever figured out that *should* be complemented.



Decimal System Unsigned	Decimal System Signed 2-Compl	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	-8	1000
9	-7	1001
10	-6	1010
11	-5	1011
12	-4	1100
13	-3	1101
14	-2	1110
15	-1	1111

Bright Idea

If you’re doing the same thing many times, do it fast! This is called `optimizing the common case`. Great advice, except when it isn’t.

“Okay, but if computers think in binary numbers, then how can you use letters?”