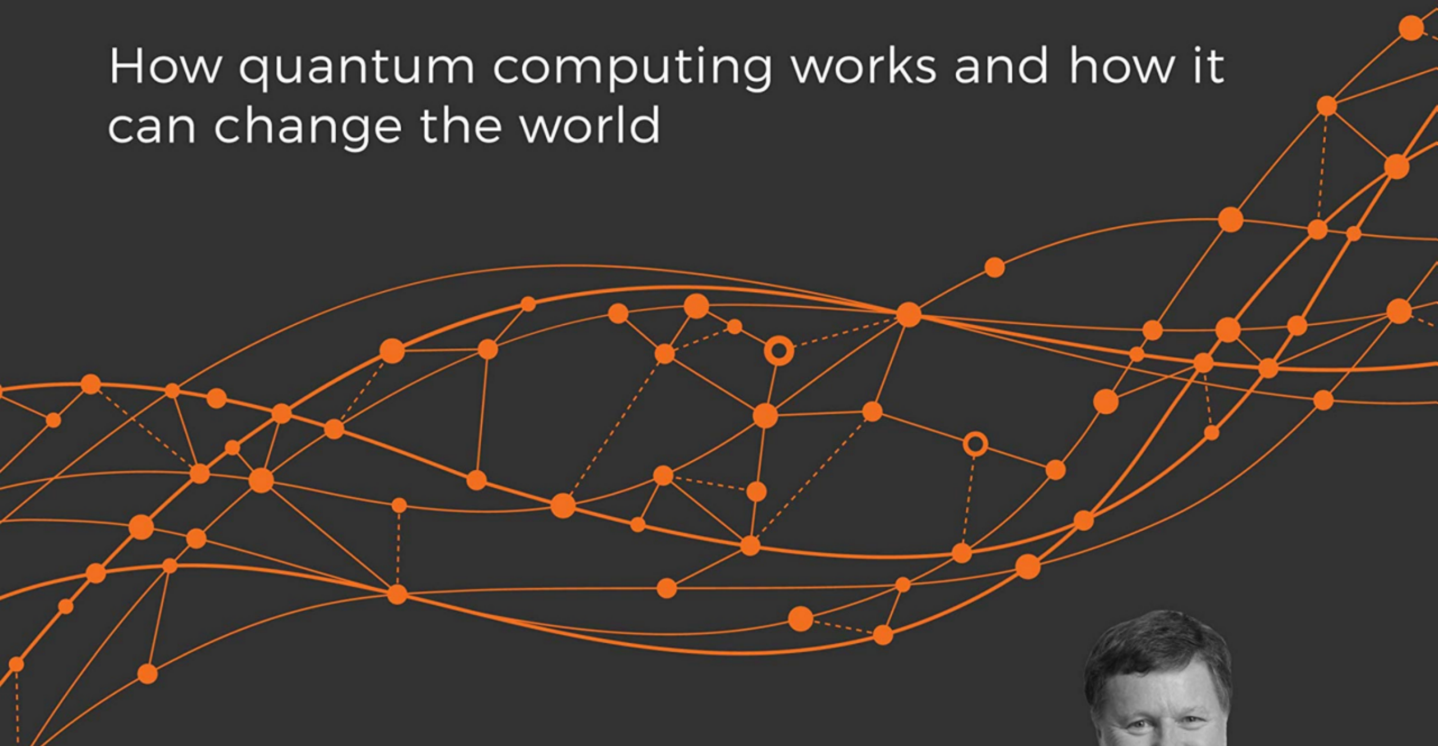
EXPERT INSIGHT

Dancing with Qubits

How quantum computing works and how it can change the world



Robert S. Susor

Packt>

Dancing with Qubits

How quantum computing works and how it can change the world

Robert S. Sutor

Packt
BIRMINGHAM – MUMBAI

Dancing with Qubits

Producer: Andrew Waldron
Development Editor: Ian Hough
Peer Review Acquisitions: Suresh Jain
Project Editor: Tom Jacob
Proof Reading: Safis Editing
Cover Designer: Sandip Tadge

First published: November 2019

Production reference: 2230120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-83882-736-6

www.packt.com

Contents

List of Figures

Preface

Everything we call real is made of things
that cannot be regarded as real.

Niels Bohr [1]

When most people think about computers, they think about laptops or maybe even the bigger machines like the servers that power the web, the Internet, and the cloud. If you look around, though, you may start seeing computers in other places. Modern cars, for example, have anywhere from around 20 computers to more than 100 to control all the systems that allow you to move, brake, monitor the air conditioning, and control the entertainment system.

The smartphone is the computer many people use more than anything else in a typical day. A modern phone has a 64-bit processor in it, whatever a "64-bit processor" is. The amount of memory used for running all those apps might be 3Gb, which means 3 gigabytes. What's a "giga" and what is a byte?

All these computers are called *classical computers* and the original ideas for them go back to the 1940s. Sounding more scientific, we say these computers have a *von Neumann architecture*, named after the mathematician and physicist John von Neumann.

It's not the 1940s anymore, obviously, but more than seventy years later we still have the modern versions of these machines in so many parts of our lives. Through the years, the "thinking" components, the processors, have gotten faster and faster. The amount of memory has also gotten larger so we can run more—and bigger—apps that do some pretty sophisticated things. The improvements in graphics processors have given us better and better games. The amount of storage has skyrocketed in the last couple of decades, so we can have more and more apps and games and photos and videos on devices we carry around with us. When it comes to these classical computers and the way they have developed, "more is better."

We can say similar things about the computer servers that run businesses and the Internet around the world. Do you store your photos in the cloud? Where is that exactly? How many photos can you keep there and how much does it cost? How quickly can your photos and all the other data you need move back and forth to that nebulous place?

It's remarkable, all this computer power. It seems like every generation of computers will continue to get faster and faster and be able to do more and more for us. There's no end in sight for how powerful these small and large machines will get to entertain us, connect us to our friends and family, and solve the important problems in the world.

Except . . . that's false.

While there will continue to be some improvements, we will not see anything like the doubling in processor power every two years that happened starting in the mid-1960s. This doubling went by the name of *Moore's Law* and went something like "every two years processors will get twice as fast, half as large, and use half as much energy."

These proportions like "double" and "half" are approximate, but physicists and engineers really did make extraordinary progress for many years. That's why you can have a computer in a watch on your wrist that is more powerful than a system that took up an entire room forty years ago.

A key problem is the part where I said processors will get half as large. We can't keep making transistors and circuits smaller and smaller indefinitely. We'll start to get so small that we approach the atomic level. The electronics will get so crowded that when we try to tell part of a processor to do something a nearby component will also get affected.

There's another deeper and more fundamental question. Just because we created an architecture over seventy years ago and have vastly improved it, does that mean all kinds of problems can eventually be successfully tackled by computers using that design? Put another way, why do we think the kinds of computers we have now might eventually be suitable for solving every possible problem? Will "more is better" run out of steam if we keep to the same kind of computer technology? Is there something wrong or limiting about our way of computing that will prevent our making the progress we need or desire?

Depending on the kind of problem you are considering, it's reasonable to think the answer to the last question if somewhere between "probably" and "yes."

That's depressing. Well, it's only depressing if we can't come up with one or more new types of computers that have a chance of breaking through the limitations.

That's what this book is about. Quantum computing as an idea goes back to at least the early 1980s. It uses the principles of quantum mechanics to provide an entirely new kind of computer architecture. Quantum mechanics in turn goes back to around 1900 but especially to the 1920s when physicists started noticing that experimental results were not matching what theories predicted.

However, this is not a book about quantum mechanics. Since 2016, tens of thousands of users have been able to use quantum computing hardware via the cloud, what we call quantum cloud services. People have started programming these new computers even though the way you do it is unlike anything done on a classical computer.

Why have so many people been drawn to quantum computing? I'm sure part of it is curiosity. There's also the science fiction angle: the word "quantum" gets tossed around enough in sci-fi movies that viewers wonder if there is any substance to the idea.

Once we get past the idea that quantum computing is new and intriguing, it's good to ask "ok, but what is it really good for?" and "when and how will it make a difference in my life?" I discuss the use cases experts think are most tractable over the next few years and decades.

It's time to learn about quantum computing. It's time to stop thinking classically and to start thinking *quantumly,* though I'm pretty sure that's not really a word!

For whom did I write this book?

This book is for anyone who has a very healthy interest in mathematics and wants to start learning about the physics, computer science, and engineering of quantum computing. I review the basic math, but things move quickly so we can dive deeply into an exposition of how to work with qubits and quantum algorithms.

While this book contains a lot of math, it is not of the definition-theorem-proof variety. I'm more interested in presenting the topics to give you insight on the relationships between the ideas than I am in giving you a strictly formal development of all results.

Another goal of mine is to prepare you to read much more advanced texts and articles on the subject, perhaps returning here to understand some core topic. You do not need to be a physicist to read this book, nor do you need to understand quantum mechanics beforehand.

At several places in the book I give some code examples using Python 3. Consider these to be extra and not required, but if you do know Python they may help in your understanding.

Many of the examples in this book come from the IBM Q quantum computing system. I was an IBM Q executive team member during the time I developed this content.

What does this book cover?

Before we jump into understanding how quantum computing works from the ground up, we need to take a little time to see how things are done classically. In fact, this is not only for the sake of comparison. The future, I believe, will be a hybrid of classical and quantum computers.

The best way to learn about something is start with basic principles and then work your way up. That way you know how to reason about it and don't rely on rote memorization or faulty analogies.

1 – Why Quantum Computing?

In the first chapter we ask the most basic question that applies to this book: why quantum computing? Why do we care? In what ways will our lives change? What are the use cases to which we hope to apply quantum computing and see a significant improvement? What do we even mean by "significant improvement"?

I – Foundations

The first full part covers the mathematics you need to understand the concepts of quantum computing. While we will ultimately be operating in very large dimensions and using complex numbers, there's a lot of insight you can gain from what happens in traditional 2D and 3D.

2 – They're Not Old, They're Classics

Classical computers are pervasive but relatively few people know what's inside them and how they work. To contrast them later with quantum computers, we look at the basics along with the reasons why they have problems doing some kinds of calculations. I introduce the simple notion of a bit, a single 0 or 1, but show that working with many bits can eventually give you all the software you use today.

3 – More Numbers than You Can Imagine

The numbers people use every day are called real numbers. Included in these are integers, rational numbers, and irrational numbers. There are other kinds of numbers, though, and structures that have many of the same algebraic properties. We look at these to lay the groundwork to understand the "compute" part of what a quantum computer does.

4 – Planes and Circles and Spheres, Oh My

From algebra we move to geometry and relate the two. What is a circle, really, and what does it have in common with a sphere when we move from two to three dimensions? Trigonometry becomes more obvious, though that is not a legally binding statement. What you thought of as a plane becomes the basis for understanding complex numbers, which are key to the definition of quantum bits, usually known as *qubits*.

5 – Dimensions

After laying the algebraic and geometric groundwork, we move beyond the familiar two- and three-dimensional world. Vector spaces generalize to many dimensions and are essential for understanding the exponential power that quantum computers can harness. What can you do when you are working in many dimensions and how should you think about such operations? This extra elbow room comes into play when we consider how quantum computing might augment AI.

6 – What Do You Mean "Probably"?

"God does not play dice with the universe," said Albert Einstein.

This was not a religious statement but rather an expression of his lack of comfort with the idea that randomness and probability play a role in how nature operates. Well, he didn't get that quite right. Quantum mechanics, the deep and often mysterious part of physics on which quantum computing is based, very much has probability at its core. Therefore, we cover the fundamentals of probability to aid your understanding of quantum processes and behavior.

II – Quantum Computing

The next part is the core of how quantum computing really works. We look at quantum bits—qubits—singly and together, and then create circuits that implement algorithms. Much of this is the ideal case when we have perfect fault-tolerant qubits. When we really create quantum computers, we must deal with the physical realities of noise and the need to reduce errors.

7 – One Qubit

At this point we are finally able to talk about qubits in a nontrivial manner. We look at both the vector and Bloch sphere representations of the quantum states of qubits. We define superposition, which explains the common cliché about a qubit being "zero and one at the same time."

8 – Two Qubits, Three

With two qubits we need more math, and so we introduce the notion of the tensor product, which allows us to explain entanglement. Entanglement, which Einstein called "spooky action at a distance," tightly correlates two qubits so that they no longer act independently. With superposition, entanglement gives rise to the very large spaces in which quantum computations can operate.

9 – Wiring Up the Circuits

Given a set of qubits, how do you manipulate them to solve problems or perform calculations? The answer is you build circuits for them out of gates that correspond to reversible operations. For now, think about the classical term "circuit board." I use the quantum analog of circuits to implement algorithms, the recipes computers use for accomplishing tasks.

10 – From Circuits to Algorithms

With several simple algorithms discussed and understood, we next turn to more complicated ones that fit together to give us Peter Shor's 1995 fast integer factoring algorithm. The math is more extensive in this chapter, but we have everything we need from previous discussions.

11 – Getting Physical

When you build a physical qubit, it doesn't behave exactly like the math and textbooks say it should. There are errors, and they may come from noise in the environment of the quantum system. I don't mean someone yelling or playing loud music, I mean fluctuating temperatures, radiation, vibration, and so on. We look at several factors you must consider when you build a quantum computer, introduce Quantum Volume as a whole-system metric of the performance of your system, and conclude with a discussion of the most famous quantum feline.

This book concludes with a chapter that moves beyond today.

12 – Questions about the Future

If I were to say, "in ten years I think quantum computing will be able to do . . . ," I would also need to describe the three or four major scientific breakthroughs that need to happen before then. I break down the different areas in which we're trying to innovate in the science and engineering of quantum computing and explain why. I also give you some guiding principles to distinguish hype from reality. All this is expressed in terms of motivating questions.

References

[1] Karen Barad. *Meeting the Universe Halfway. Quantum Physics and the Entanglement of Matter and Meaning*. 2nd ed. Duke University Press Books, 2007.

What conventions are used in this book?

When I want to highlight something important that you should especially remember, I use this kind of box:

This is very important.

This book does not have exercises but it does have questions. Some are answered in the text and others are left for you as thought experiments. Try to work them out as you go along. They are numbered within chapters.

Question 0.0.1

Why do you ask so many questions?

Code samples and output are presented to give you an idea about how to use a modern programming language, Python 3, to experiment with basic ideas in quantum computing.

```
def obligatoryFunction():
    print("Hello quantum world!")

obligatoryFunction()

Hello quantum world!
```

Numbers in brackets (for example, [1]) are references to additional reading materials. They are listed at the end of each chapter in which the bracketed number appears.

To learn more

Here is a place where you might see a reference to learn more about some topic. [1]

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, http://www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit http://authors.packtpub.com.

. . .

Now let's get started by seeing why we should look at quantum computing systems to try to solve problems that are intractable with classical systems.

1

Why Quantum Computing?

Nature isn't classical, dammit,
and if you want to make a simulation of nature,
you'd better make it quantum mechanical.

Richard Feynman [5]

In his 1982 paper "Simulating Physics with Computers," Richard Feynman, 1965 Nobel Laureate in Physics, said he wanted to "talk about the possibility that there is to be an *exact* simulation, that the computer will do *exactly* the same as nature." He then went on to make the statement above, asserting that nature doesn't especially make itself amenable for computation via classical binary computers.

In this chapter we begin to explore how quantum computing is different from classical computing. Classical computing is what drives smartphones, laptops, Internet servers, mainframes, high performance computers, and even the processors in automobiles.

We examine several use cases where quantum computing may someday help us solve problems that are today intractable using classical methods on classical computers. This is to motivate you to learn about the underpinnings and details of quantum computers I discuss throughout the book.

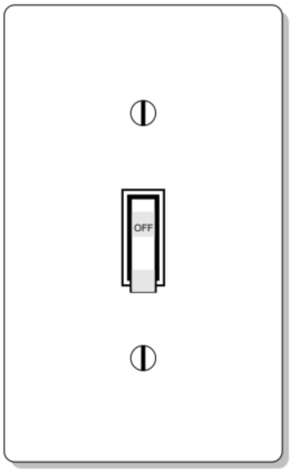
No single book on this topic can be complete. The technology and potential use cases are moving targets as we innovate and create better hardware and software. My goal here is to prepare you to delve more deeply into the science, coding, and applications of quantum computing.

Topics covered in this chapter

1.1 The mysterious quantum bit

Suppose I am standing in a room with a single overhead light and a switch that turns the light on or off. This is just a normal switch, and so I can't dim the light. It is either fully on or fully off. I can change it at will, but this is the only thing I can do to it. There is a single door to the room and no windows. When the door is closed I cannot see any light.

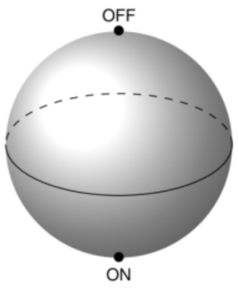
I can stay in the room or I may leave it. The light is always on or off based on the position of the switch.



Now I'm going to do some rewiring. I'm replacing the switch with one that is in another part of the building. I can't see the light at all but, once again, its being on or off is determined solely by the two positions of the switch.

If I walk to the room with the light and open the door, I can see whether it is lit or dark. I can walk in and out of the room as many times as I want and the status of the light is still determined by that remote switch being on or off. This is a "classical" light.

Now let's imagine a *quantum* light and switch, which I'll call a "qu-light" and "qu-switch," respectively.



When I walk into the room with the qu-light it is always on or off, just like before. The qu-switch is unusual in that it is shaped like a sphere with the topmost point (the "north pole") being OFF and the bottommost (the "south pole") being ON. There is a line etched around the middle.

The interesting part happens when I cannot see the qu-light, when I am in the other part of the building with the qu-switch.

I control the qu-switch by placing my index finger on the qu-switch sphere. If I place my finger on the north pole, the qu-light is definitely off. If I put it on the south, the qu-light is definitely on. You can go into the room and check. You will always get these results.

If I move my finger anywhere else on the qu-switch sphere, the qu-light may be on or off when you check. If you do not check, the qu-light is in an indeterminate state. It is not dimmed, it is not on or off, it just exists with some probability of being on or off when seen. This is unusual!

The moment you open the door and see the qu-light, the indeterminacy is removed. It will be on or off. Moreover, if I had my finger on the qu-switch, the finger would be forced to one or other of the poles corresponding to the state of the qu-light when it was seen.

The act of observing the qu-light forced it into either the on or off state. I don't have to see the qu-light fixture itself. If I open the door a tiny bit, enough to see if any light is shining or not, that is enough.

If I place a video camera in the room with the qu-light and watch it when I try to place my finger on the qu-switch, it behaves just like a normal switch. I will be prevented from touching the qu-switch at anywhere other than the top or bottom. Since I'm making up this example, assume some sort of force field keeps me away from anywhere but the poles!

If you or I are not observing the qu-light in any way, does it make a difference where I touch the qu-switch? Will touching it in the northern or southern hemisphere influence whether it will be on or off when I observe the qu-light?

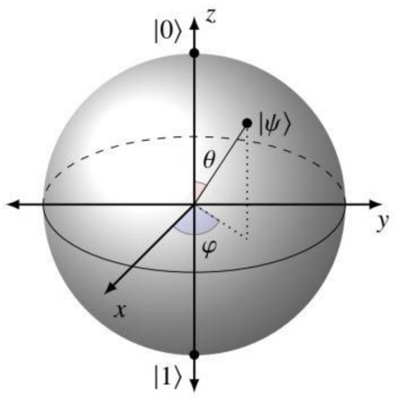
Yes. Touching it closer to the north pole or the south pole will make the probability of the qu-light being off or on, respectively, be higher. If I put my finger on the circle between the poles, the equator, the probability of the light being on or off will be exactly 50 – 50.

What I just described is called a *two-state quantum system*. When it is not being observed, the qu-light is in a *superposition* of being on and off. We explore superposition in section 7.1.

While this may seem bizarre, evidently nature really works this way. Electrons have a property called "spin" and with this they are two-state quantum systems. The photons that make up light itself are two-state quantum systems. We return to this in section 11.3 when we look at polarization (as in Polaroid® sunglasses).

More to the point of this book, however, a *quantum bit*, more commonly known as a *qubit*, is a two-state quantum system. It extends and complements the classical computing notion of bit, which can only be 0 or 1. The qubit is the basic information unit in quantum computing.

This book is about how we manipulate qubits to solve problems that currently appear to be intractable using just classical computing. It seems that just sticking to 0 or 1 will not be sufficient to solve some problems that would otherwise need impractical amounts of time or memory.



With a qubit, we replace the terminology of on or off, 1 or 0, with $|1\rangle$ and $|0\rangle$, respectively. Instead of qu-lights, it's qubits from now on.

In the diagram on the right, the position of your finger on the qu-switch is now indicated by two angles, θ and φ. The picture itself is called a Bloch sphere and is a standard representation of a qubit, as we shall see in section 7.5.

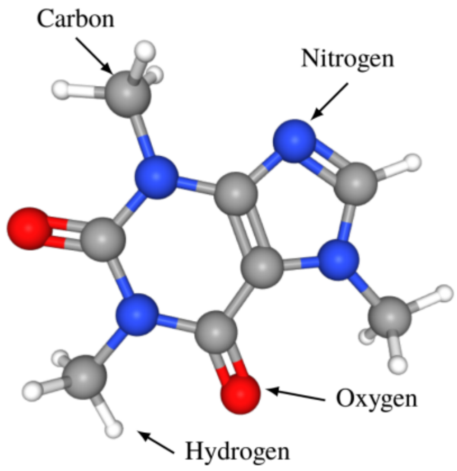
1.2 I'm awake!

What if we could do chemistry inside a computer instead of in a test tube or beaker in the laboratory? What if running a new experiment was as simple as running an app and having it complete in a few seconds?

For this to really work, we would want it to happen with full *fidelity*. The atoms and molecules as modeled in the computer should behave **exactly** like they do in the test tube. The chemical reactions that happen in the physical world would have precise computational analogs. We would need a fully faithful simulation.

If we could do this at scale, we might be able to compute the molecules we want and need. These might be for new materials for shampoos or even alloys for cars and airplanes. Perhaps we could more efficiently discover medicines that are customized to your exact physiology. Maybe we could get better insight into how proteins fold, thereby understanding their function, and possibly creating custom enzymes to positively change our body chemistry.

Is this plausible? We have massive supercomputers that can run all kinds of simulations. Can we model molecules in the above ways today?



Let's start with $C_8H_{10}N_4O_2$ – 1,3,7-Trimethylxanthine. This is a very fancy name for a molecule which millions of people around the world enjoy every day: **caffeine**. An 8 ounce cup of coffee contains approximately 95 mg of caffeine, and this translates to roughly 2.95×10^{20} molecules. Written out, this is

$$295{,}000{,}000{,}000{,}000{,}000{,}000 \text{ molecules.}$$

A 12 ounce can of a popular cola drink has 32 mg of caffeine, the diet version has 42 mg, and energy drinks often have about 77 mg. [11]

Question 1.2.1

How many molecules of caffeine do you consume a day?

These numbers are large because we are counting physical objects in our universe, which we know is very big. Scientists estimate, for example, that there are between 10^{49} and 10^{50} atoms in our planet alone. [4]

To put these values in context, one thousand = 10^3, one million = 10^6, one billion = 10^9, and so on. A gigabyte of storage is one billion bytes, and a terabyte is 10^{12} bytes.

Getting back to the question I posed at the beginning of this section, can we model caffeine exactly in a computer? We don't have to model the huge number of caffeine molecules in a cup of coffee, but can we fully represent a single molecule at a single instant?

Caffeine is a small molecule and contains protons, neutrons, and electrons. In particular, if we just look at the energy configuration that determines the structure of the molecule and the bonds that hold it all together, the amount of information to describe this is staggering. In particular, the number of bits, the 0s and 1s, needed is approximately 10^{48}:

$$10,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000.$$

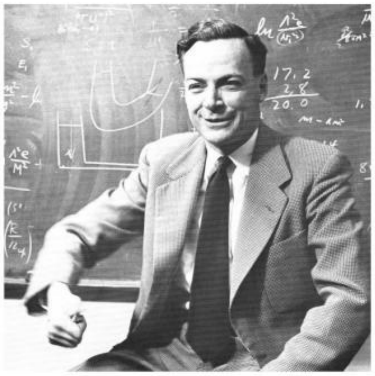
From what I said above, this is comparable to 1% to 10% of the number of atoms on the Earth.

This is just one molecule! Yet somehow nature manages to deal quite effectively with all this information. It handles the single caffeine molecule, to all those in your coffee, tea, or soft drink, to every other molecule that makes up you and the world around you.

How does it do this? We don't know! Of course, there are theories and these live at the intersection of physics and philosophy. We do not need to understand it fully to try to harness its capabilities.

We have no hope of providing enough traditional storage to hold this much information. Our dream of exact representation appears to be dashed. This is what Richard Feynman meant in his quote at the beginning of this chapter: "Nature isn't classical."

However, 160 qubits (quantum bits) could hold $2^{160} \approx 1.46 \times 10^{48}$ bits while the qubits were involved in computation. To be clear, I'm not saying how we would get all the data into those qubits and I'm also not saying how many more we would need to do something interesting with the information. It does give us hope, however.


Richard Feynman at the California Institute of Technology in 1959. Photo is in the public domain. Ⓟ

In the classical case, we will never fully represent the caffeine molecule. In the future, with enough very high quality qubits in a powerful enough quantum computing system, we may be able to perform chemistry in a computer.

To learn more

Quantum chemistry is not an area of science in which you can say a few words and easily make clear how quantum computers might eventually be used to compute molecular properties and protein folding configurations, for example. Nevertheless, the caffeine example above is an example of *quantum simulation*.

For an excellent survey of the history and state of the art of quantum computing applied to chemistry as of 2019, see Cao et al. [2] For the specific problem of understanding how to scale quantum simulations of molecules and the crossover from High Performance Computers (HPC), see Kandala et al. [10]

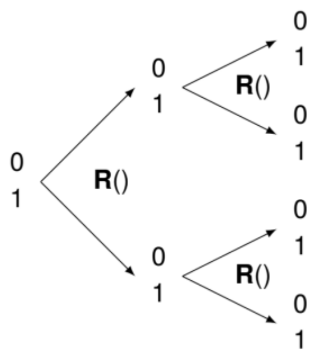
1.3 Why quantum computing is different

I can write a little app on a classical computer that can simulate a coin flip. This might be for my phone or laptop.

Instead of heads or tails, let's use 1 and 0. The routine, which I call **R**, starts with one of those values and randomly returns one or the other. That is, 50% of the time it returns 1 and 50% of the time it returns 0. We have no knowledge whatsoever of how **R** does what it does. When you see "**R**," think "random."

This is called a "fair flip." It is not weighted to slightly prefer one result or the other. Whether we can produce a truly random result on a classical computer is another question. Let's assume our app is fair.

If I apply **R** to 1, half the time I expect that same value and the other half 0. The same is true if I apply **R** to 0. I'll call these applications **R**(1) and **R**(0), respectively.



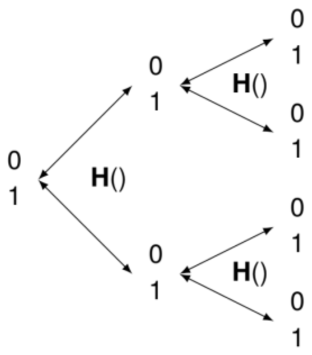
If I look at the result of **R**(1) or **R**(0), there is no way to tell if I started with 1 or 0. This is just as in a secret coin flip where I can't tell whether I began with heads or tails just by looking at how the coin has landed. By "secret coin flip," I mean that someone else does it and I can see the result, but I have no knowledge of the mechanics of the flip itself or the starting state of the coin.

If **R**(1) and **R**(0) are randomly 1 and 0, what happens when I apply **R** twice?

I write this as **R**(**R**(1)) and **R**(**R**(0)). It's the same answer: random result with an equal split. The same thing happens no matter how many times we apply **R**. The result is random and we can't reverse things to learn the initial value. In the language of section 4.1, **R** is not *invertible*.

Now for the quantum version. Instead of **R**, I use **H**, which we learn about in section 7.6. It too returns 0 or 1 with equal chance but it has two interesting properties:

1. It is reversible. Though it produces a random 1 or 0 starting from either of them, we can always go back and see the value with which we began.
2. It is its own reverse (or *inverse*) operation. Applying it two times in a row is the same as having done nothing at all.



There is a catch, though. You are not allowed to look at the result of what **H** does if you want to reverse its effect.

If you apply **H** to 0 or 1, peek at the result, and apply **H** again to that, it is the same as if you had used **R**. If you observe what is going on in the quantum case at the wrong time, you are right back at strictly classical behavior.

To summarize using the coin language: if you flip a quantum coin and then **don't look at it**, flipping it again will yield the heads or tails with which you started. If you do look, you get classical randomness.

Question 1.3.1

Compare this behavior with that of the qu-switch and qu-light in section 1.1.

A second area where quantum is different is in how we can work with simultaneous values. Your phone or laptop uses bytes as the individual units of memory or storage. That's where we get phrases like "megabyte," which means one million bytes of information.

A byte is further broken down into eight bits, which we've see before. Each bit can be 0 or 1. Doing the math, each byte can represent $2^8 = 256$ different numbers composed of eight 0s or 1s, but it can only hold *one value at a time*.

Eight qubits can represent all 256 values *at the same time*.

This is through superposition, but also through *entanglement*, the way we can tightly tie together the behavior of two or more qubits. This is what gives us the (literally) exponential growth in the amount of working memory that we saw with a quantum representation of caffeine in section 1.2. We explore entanglement in section 8.2.

1.4 Applications to artificial intelligence

Artificial intelligence and one of its subsets, machine learning, are extremely broad collections of data-driven techniques and models. They are used to help find patterns in information, learn from the information, and automatically perform more "intelligently." They also give humans help and insight that might have been difficult to get otherwise.

Here is a way to start thinking about how quantum computing might be applicable to large, complicated, computation-intensive systems of processes such as those found in AI and elsewhere. These three cases are in some sense the "small, medium, and large" ways quantum computing might complement classical techniques:

1. There is a single mathematical computation somewhere in the middle of a software component that might be sped up via a quantum algorithm.
2. There is a well described component of a classical process that could be replaced with a quantum version.
3. There is a way to avoid the use of some classical components entirely in the traditional method because of quantum, or the entire classical algorithm can be replaced by a much faster or more effective quantum alternative.

As I write this, quantum computers are not "big data" machines. This means you cannot take millions of records of information and provide them as input to a quantum calculation. Instead, quantum may be able to help where the number of inputs are modest but the computations "blow up" as you start examining relationships or dependencies in the data. Quantum, with its exponentially growing working memory, as we saw in the caffeine example in section 1.2, may be able to control and work with the blow up. (See section 2.7 for a discussion of exponential growth.)

In the future, however, quantum computers may be able to input, output, and process much more data. Even if it is just theoretical now, it makes sense to ask if there are quantum algorithms that can be useful in AI someday.

Let's look at some data. I'm a big baseball fan, and baseball has a lot of statistics associated with it. The analysis of this even has its own name: "sabermetrics."

Year	**GP**	**AB**	**R**	**H**	**2B**	**3B**	**HR**	**RBI**	**BB**	**SO**
2019	136	470	105	136	27	2	41	101	110	111
2018	162	587	94	156	25	1	46	114	74	173
2017	152	542	73	132	24	0	29	92	41	145
2016	140	490	84	123	26	5	27	70	50	109
2015	162	634	66	172	32	4	25	83	26	108
2014	148	545	110	153	35	1	29	79	74	144

where

GP = Games Played	**3B** = 3 Base hits (triples)
AB = At Bats	**HR** = Home Runs
R = Runs scored	**RBI** = Runs Batted In
H = Hits	**BB** = Bases on Balls (walks)
2B = 2 Base hits (doubles)	**SO** = Strike Outs

Figure 1.1: Baseball player statistics by year

Suppose I have a table of statistics for a baseball player given by year as shown in Figure 1.1. We can make this look more mathematical by creating a matrix of the same data.

$$\begin{bmatrix} 2019 & 136 & 470 & 105 & 136 & 27 & 2 & 41 & 101 & 110 & 111 \\ 2018 & 162 & 587 & 94 & 156 & 25 & 1 & 46 & 114 & 74 & 173 \\ 2017 & 152 & 542 & 73 & 132 & 24 & 0 & 29 & 92 & 41 & 145 \\ 2016 & 140 & 490 & 84 & 123 & 26 & 5 & 27 & 70 & 50 & 109 \\ 2015 & 162 & 634 & 66 & 172 & 32 & 4 & 25 & 83 & 26 & 108 \\ 2014 & 148 & 545 & 110 & 153 & 35 & 1 & 29 & 79 & 74 & 144 \end{bmatrix}$$

Given such information, we can manipulate it using machine learning techniques to make predictions about the player's future performance or even how other similar players may do. These techniques make use of the matrix operations we discuss in chapter 5.

There are 30 teams in Major League Baseball in the United States. With their training and feeder "minor league" teams, each major league team may each have more than 400 players throughout their systems. That would give us over 12,000 players, each with their complete player histories. There are more statistics than I have listed, so we can easily get greater than 100,000 values in our matrix.

In the area of entertainment, it's hard to make an estimate of how many movies exist, but it is well above 100,000. For each movie, we can list features such as whether it is a comedy or a drama or a romance or an action film, who each of the actors are, who each of the directorial and production staff are, geographic locations shown in the fim, languages used, and so on. There are hundreds of such features and *millions of people who have watched the films!*

For each person, we can also add features such as whether they like or dislike a kind of movie, actor, scene location, or director. Using all this information, which film should I recommend to you on a Saturday night in December based on what you and people similar to you like?

Think of each feature or each baseball player or film as a dimension. While you may think of two and three dimensions in nature, in AI we might have thousands or millions of dimensions.

Matrices as above for AI can grow to millions of rows and entries. How can we make sense of them to get insights and see patterns? Aside from manipulating that much information, can we even eventually do the math on classical computers quickly and accurately enough?

While it was originally thought that quantum algorithms might offer exponential improvements of such classical recommender systems, a 2019 "quantum-inspired algorithm" by Ewin Tang showed a classical method to gain such a huge improvement. [17] An example of being exponentially faster is doing something in 6 days instead of $10^6 = 1$ million days. That's approximately 2,740 years.

Tang's work is a fascinating example of the interplay of progress in both classical and quantum algorithms. People who develop algorithms for classical computing look to quantum computing, and vice versa. Also, any particular solution to a problem many include classical and quantum components.

Nevertheless, many believe that quantum computing will show very large improvements for some matrix computations. One such example is the **HHL** algorithm, whose abbreviation comes from the first letters of the last names of its authors, Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. This is also an example of case number 1 above.

Algorithms such as these may find use in fields as diverse as economics and computational fluid dynamics. They also place requirements on the structure and density of the data and may use properties such as the condition number we discuss in subsection 5.7.6.

To learn more

When you complete this book you will be equipped to read the original paper describing the **HHL** algorithm and more recent surveys about how to apply quantum computing to linear algebraic problems. [7]

An important problem in machine learning is classification. In its simplest form, a *binary classifier* separates items into one of two categories, or buckets. Depending on the definitions of the categories, it may be more or less easy to do the classification.

Examples of binary categories include:

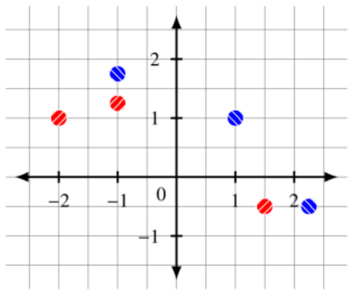
- book you like **or** book you don't like
- comedy movie **or** dramatic movie
- gluten-free **or** not gluten-free
- fish dish **or** chicken dish
- UK football team **or** Spanish football team
- hot sauce **or** very hot sauce
- cotton shirt **or** permanent press shirt
- open source **or** proprietary
- spam email **or** valid email
- American League baseball team **or** National League team

The second example of distinguishing between comedies and dramas may not be well designed since there are movies that are both.

Mathematically, we can imagine taking some data as input and classifying it as either +1 or −1. We take a reasonably large set of data and label it by hand as either being a +1 or −1. We then *learn* from this *training set* how to classify future data.

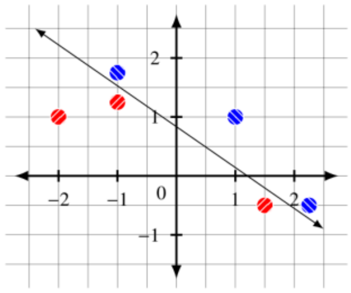
Machine learning binary classification algorithms include random forest, k-nearest neighbor, decision tree, neural networks, naive Bayes classifiers, and support vector machines (SVMs).

In the training phase, we are given a list of pre-classified objects (books, movies, proteins, operating systems, baseball teams, etc.). We then use the above algorithms to learn how to put a new object in one bucket or another.



The SVM is a straightforward approach with a clear mathematical description. In the two-dimensional case, we try to draw a line that separates the objects (represented by points in the plot to the right) into one category or the other.

The line should maximize the gap between the sets of objects.



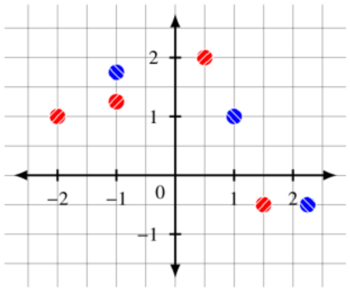
On the left is an example of a line that separates the red points below from the blue points above the line.

Given a new point, we plot it and determine whether it is above or below the line. That will classify it as blue or red, respectively.

Suppose we know that the point is correctly classified with those above the line. We accept that and move on.

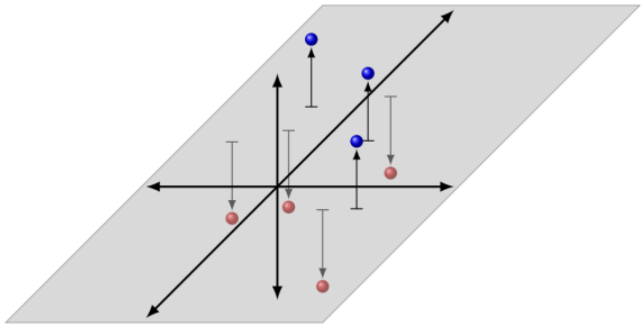
If the point is misclassified, we add the point to the training set and try to compute a new and better line. This may not be possible.

In the plot to the right, I added a new red point above the line close to 2 on the vertical axis. With this extra point, there is no line we can compute to separate the points.



Had we represented the objects in three dimensions, we would try to find a plane that separated the points with maximum gap. We would need to compute some new amount that the points are above or below the plane. In geometric terms, if we are given x and y only, we somehow need to compute a z to work in that third dimension.

For a representation using n dimensions, we try to compute an $n-1$ separating *hyperplane*. We look at two and three dimensions in chapter 4 and the general case in chapter 5.



In this three-dimensional plot, I take the same values from the last two-dimensional version and lay the coordinate plane flat. I then add a vertical dimension. I push the red points below the plane and the blue ones above. With this construction, the coordinate plane itself separates the values.

While we can't separate the points in two dimensions, we can in three dimensions. This kind of mapping into a higher dimension is called the *kernel trick*. While the coordinate plane in this case might not be the ideal separating hyperplane, it gives you an idea of what we are trying to accomplish. The benefit of *kernel functions* (as part of the similarly named "trick") is that we can do far fewer explicit geometric computations than you might expect in these higher dimensional spaces.

It's worth mentioning now that we don't need to try quantum methods on small problems that are handled quite well using traditional means. We won't see any kind of quantum advantage until the problems are big enough to overcome the quantum circuit overhead versus classical circuits. Also, if we come up with a quantum approach that can be simulated easily on a classical computer, we don't really need a quantum computer.

A quantum computer with 1 qubit provides us with a two-dimensional working space. Every time we add a qubit, we double the number of dimensions. This is due to the properties of superposition and entanglement that I introduce in chapter 7. For 10 qubits, we get $2^{10} = 1024$ dimensions. Similarly, for 50 qubits we get $2^{50} = 1,125,899,906,842,624$ dimensions.

Remember all those dimensions for the features and baseball players and films? We want to use a sufficiently large quantum computer to do the AI calculations in a *quantum feature space*. This is the main point: handle the extremely large number of dimensions coming out of the data in a large quantum feature space.

There is a quantum approach that can generate the separating hyperplane in the quantum feature space. There is another that skips the hyperplane step and produces a highly accurate classifying kernel function. As the ability to entangle more qubits increases, the successful classification rate improves as well. [8] This is an active area of research: how can we use entanglement, which does not exist classically, to find new or better patterns than we can do with strictly traditional methods?

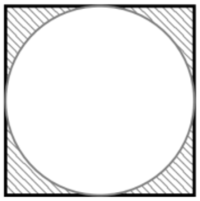
To learn more

There are an increasing number of research papers being written connecting quantum computing with machine learning and other AI techniques. The results are somewhat fragmented. The best book pulling together the state of the art is Wittek. [19].

I do warn you again that quantum computers cannot process much data now!

For an advanced application of machine learning for quantum computing and chemistry, see Torial et al. [18]

1.5 Applications to financial services



Suppose we have a circle of radius 1 inscribed in a square, which therefore has sides of length 2 and area $4 = 2 \times 2$. What is the area A of the circle?

Before you try to remember your geometric area formulas, let's compute it a different way using ratios and some experiments.

Suppose we drop some number N of coins onto the square and count how many of them have their centers on or inside the circle. If C is this number, then

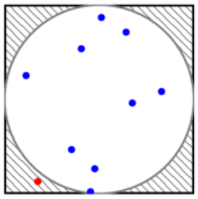
$$\frac{\text{area of the circle}}{\text{area of the enclosing square}} = \frac{A}{4} \approx \frac{C}{N} = \frac{\text{number of coins that land in the circle}}{\text{total number of coins}}.$$

So $A \approx \frac{4C}{N}$.

There is randomness involved here: it's possible they all land inside the circle or, less likely, outside the circle. For $N = 1$, we do not get an accurate estimate of A at all because $\frac{C}{N}$ can only be 0 or 1.

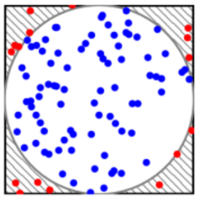
Question 1.5.1

If $N = 2$, what are the possible estimates of A? What about if $N = 3$?

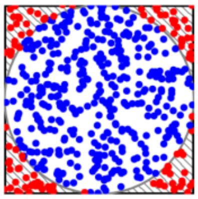


Clearly we will get a better estimate for A if we choose N large.

Using Python and its random number generator, I created 10 points whose centers lie inside the square. The plot on the right shows where they landed. In this case, $C = 9$ and so $A \approx 3.6$.



For $N = 100$ we get a more interesting plot with $C = 84$ and $A \approx 3.36$. Remember, if different random numbers had been generated then this number would be different.



The final plot is for $N = 500$. Now $C = 387$ and $A \approx 3.096$.

The real value of A is $\pi \approx 3.1415926$. This technique is called *Monte Carlo sampling* and goes back to the 1940s.

Using the same technique, here are approximations of A for increasingly large N. Remember, we are using random numbers and so these numbers will vary based on the sequence of values used.

N	10	100	1,000	10,000	100,000	1,000,000	10,000,000
A	3.6	3.36	3.148	3.1596	3.14336	3.141884	3.1414132

That's a lot of runs, the value of N, to get close to the real value of π. Nevertheless, this example demonstrates how we can use Monte Carlo sampling techniques to approximate the value of something when we may not have a formula. In this case we estimated A. For the example we ignored our knowledge that the formula for the area of a circle is πr^2, where r is the circle's radius.

In section 6.7 we work through the math and show that if we want to estimate π within 0.00001 with probability at least 99.9999%, we need $N \geq 82,863,028$. That is, we need to use more than 82 million points! So it is possible to use a Monte Carlo method here but it is not efficient.

In this example, we know the answer ahead of time by other means. If we do not know the answer and do not have a nice neat formula to compute, Monte Carlo methods can be a useful tool. However, the very large number of samples needed to get decent accuracy makes the process computationally intensive. If we can reduce the sample count significantly, we can compute a more accurate result much faster.

Given that the title of this section mentions "finance," I now note, perhaps unsurprisingly, that Monte Carlo methods are used in computational finance. The randomness we use to calculate π translates over into ideas like uncertainties. Uncertainties can then be related to probabilities, which can be used to calculate the risk and rate of return of investments.

Instead of looking at whether a point is inside or outside a circle, for the rate of return we might consider several factors that go into calculating the risk. For example,

- market size,
- share of market,
- selling price,
- fixed costs,
- operating costs,
- obsolescence,
- inflation or deflation,
- national monetary policy,
- weather, and
- political factors and election results.

For each of these or any other factor relevant to the particular investment, we quantify them and assign probabilities to the possible results. In a weighted way, we combine all possible combinations to compute risk. This is a function that we cannot calculate all at once, but we can use Monte Carlo methods to estimate. Methods similar to, but more complicated than, the circle analysis in section 6.7, give us how many samples we need to use to get a result within a desired accuracy.

In the circle example, even getting reasonable accuracy can require tens of millions of samples. For an investment risk analysis we might need many orders of magnitude greater. So what do we do?

We can and do use High Performance Computers (HPC). We can consider fewer possibilities for each factor. For example, we might vary the possible selling prices by larger amounts. We can consult better experts and get more accurate probabilities. This could increase the accuracy but not necessarily the computation time. We can take fewer samples and accept less precise results.

Or we might consider quantum variations and replacements for Monte Carlo methods. In 2015, Ashley Montanaro described a quadratic speedup using quantum computing. [12] How much improvement does this give us? Instead of the 82 million samples required for the circle calculation with the above accuracy, we could do it in something closer to 9,000 samples. ($9055 \approx \sqrt{82000000}$.)

In 2019, Stamatopoulos *et al* showed methods and considerations for pricing financial options using quantum computing systems. [16] I want to stress that to do this, we will need much larger, more accurate, and more powerful quantum computers than we have as of this writing. However, like much of the algorithmic work being done on industry use cases, we believe we are getting on the right path to solve significant problems significantly faster using quantum computation.

By using Monte Carlo methods we can vary our assumptions and do scenario analysis. If we can eventually use quantum computers to greatly reduce the number of samples, we can look at far more scenarios much faster.

To learn more

David Hertz' original 1964 paper in the Harvard Business Review is a very readable introduction to Monte Carlo methods for risk analysis without ever using the phrase "Monte Carlo." [9] A more recent paper gives more of the history of these methods and applies them to marketing analytics. [6]

My goal with this book is to give you enough of an introduction to quantum computing so that you can read industry-specific quantum use cases and research papers. For example, to see modern quantum algorithmic approaches to risk analysis, see the articles by Woerner, Egger, et al. [20] [3]. Some early results on heuristics using quantum computation for transaction settlements are covered in Braine et al. [1]

1.6 What about cryptography?

You may have seen media headlines like

Quantum Security Apocalypse!!!

Y2K??? Get ready for Q2K!!!

Quantum Computing Will Break All Internet Security!!!

These breathless announcements are meant to grab your attention and frequently contain egregious errors about quantum computing and security. Let's look at the root of the concerns and insert some reality into the discussion.

RSA is a commonly used security protocol and it works something like this:

- You want to allow others to send you secure communications. This means you give them what they need to encrypt their messages before sending. You and only you can decrypt what they then give you.
- You publish a *public key* used to encrypt these messages intended for you. Anyone who has access to the key can use it.
- There is an additional key, your *private key*. You and only you have it. With it you can decrypt and read the encrypted messages. [15]

Though I phrased this in terms of messages sent to you, the scheme is adaptable for sending transaction and purchase data across the Internet, and storing information securely in a database.

Certainly if anyone steals your private key, there is a cybersecurity emergency. Quantum computing has nothing to do with physically taking your private key or convincing you to give it to a bad person.

What if I could compute your private key from the public key?

The public key for RSA looks like a pair of numbers (e, n) where n is a very larger integer that is the product of two primes. We'll call these primes numbers p and q. For example, if $p = 982451653$ and $q = 899809343$, then $n = pq = 884019176415193979$.

Your private key looks like a pair of integers (d, n) using the very same n as in the public key. It is the d part you must really keep secret.

Here's the potential problem: if someone can quickly factor n into p and q, then they can compute d. That is, fast integer factorization leads to breaking RSA encryption.

Though multiplication is very easy and can be done using the method you learned early in your education, factoring can be very, very hard. For products of certain pairs of primes, factorization using known classical methods could take hundreds or thousands of **years**.

Given this, unless d is stolen or given away, you might feel pretty comfortable about security. Unless, that is, there is another way of factoring involving non-classical computers.

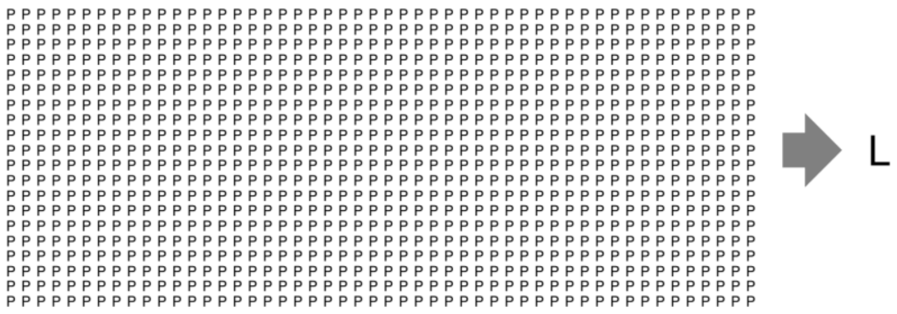
In 1995, Peter Shor published a quantum algorithm for integer factorization that is almost exponentially faster than known classical methods. We analyze Shor's algorithm in section 10.6.

This sounds like a major problem! Here is where many of the articles about quantum computing and security start to go crazy. The key question is: **how powerful, and of what quality, must a quantum computing system be in order to perform this factorization?**

As I write this, scientists and engineers are building quantum computers with double digit numbers of *physical* qubits, hoping to get to triple digits in the next few years. For example, researchers have discussed qubit counts of 20, 53, 72, and 128. (Do note there is a difference between what people say they will have versus what they really have.) A physical qubit is the hardware implementation of the *logical* qubits we start discussing in chapter 7.

Physical qubits have noise that cause errors in computation. Shor's algorithm requires fully fault-tolerant, error corrected logical qubits. This means we can detect and correct any errors that occur in the qubits. This happens today in the memory and data storage in your laptop and smartphone. We explore quantum error correction in section 11.5.

As a rule of thumb, assume it will take 1,000 very good physical qubits to make one logical qubit. This estimate varies by researcher, degree of marketing hype, and wishful thinking, but I believe 1,000 is reasonable. We discuss the relationship between the two kinds of qubits in chapter 11. In the meanwhile, we are in the Noisy Intermediate-Scale Quantum, or NISQ, era. The term NISQ was coined by physicist John Preskill in 2018. [14]



It will take many physical qubits to make one logical qubit

A further estimate is that it will take 10^8 = 100 million physical qubits to use Shor's algorithm to factor the values of n used in RSA today. That's approximately one hundred thousand logical qubits. On one hand, we have quantum computers with two or three digits worth of physical qubits. For Shor's algorithm to break RSA, we'll need eight digits worth. That's a huge difference.

These numbers may be too conservative, but I don't think by much. If anyone quotes you much smaller numbers, try to understand their motivation and what data they are using.

There's a good chance we won't get quantum computers this powerful until 2035 or much later. We may never get such powerful machines. Assuming we will, what should you do now?

First, you should start moving to so-called "post-quantum" or "quantum-proof" encryption protocols. These are being standardized at NIST, the National Institute of Standards and Technology, in the United States by an international team of researchers. These protocols can't be broken by quantum computing systems as RSA and some of the other classical protocols might be eventually.

You may think you have plenty of time to change over your transactional systems. How long will it take to do that? For financial institutions, it can take ten years or more to implement new security technology.

Of greater immediate importance is your data. Will it be a problem if someone can crack your database security in 15, 30, or 50 years? For most organizations the answer is a loud YES. Start looking at hardware and software encryption support for your data using the new post-quantum security standards now.

Finally, quantum or no quantum, if you do not have good cybersecurity and encryption strategies and implementations in place now, you are exposed. Fix them. Listen to the people who make quantum computing systems to get a good idea of if and when they might be used to break encryption schemes. All others are dealing with second- and third-hand knowledge.

To learn more

Estimates for when and if quantum computing may pose a cybersecurity threat vary significantly. Any study on the topic will necessarily need to be updated as the technology evolves. The most complete analysis as of the time this book was first published appears to be Mosca and Piani. [13]

1.7 Summary

In this first chapter we looked at what is motivating the recent interest in quantum computers. The lone 1s and 0s of classical computing bits are extended and complemented by the infinite states of qubits, also known as quantum bits. The properties of superposition and entanglement give us access to many dimensions of working memory that are unavailable to classical computers.

Industry use cases for quantum computing are nascent but the areas where experts believe it will be applicable sooner are chemistry, materials science, and financial services. AI is another area where quantum may boost performance for some kinds of calculations.

There has been confusion in traditional and social media about the interplay of security, information encryption, and quantum computing. The major areas of misunderstanding are the necessary performance requirements and the timeline.

In the next chapter, we look at classical bit-based computing to more precisely and technically explore how quantum computing may help us attack problems that are otherwise impossible today. In chapter 3 through chapter 6 we work through the mathematics necessary for you to see how quantum computing works. There is a lot to cover, but it is worth it to be able to go deeper than a merely superficial understanding of the "whats," "hows," and "whys" of quantum computing.

References

[1] Lee Braine et al. *Quantum Algorithms for Mixed Binary Optimization applied to Transaction Settlement*. 2019. URL: https://arxiv.org/abs/1910.05788.

[2] Yudong Cao et al. "Quantum Chemistry in the Age of Quantum Computing". In: *Chemical Reviews* (2019).

[3] Daniel J. Egger et al. *Credit Risk Analysis using Quantum Computers*. 2019. URL: https://arxiv.org/abs/1907.03044.

[4] FermiLab. *What is the number of atoms in the world?* 2014. URL: https://www.fnal.gov/pub/science/inquiring/questions/atoms.html.

[5] Richard P. Feynman. “Simulating Physics with Computers”. In: *International Journal of Theoretical Physics* 21.6 (June 1, 1982), pp. 467–488.

[6] Peter Furness. “Applications of Monte Carlo Simulation in marketing analytics”. In: *Journal of Direct, Data and Digital Marketing Practice* 13 (2 Oct. 27, 2011).

[7] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. “Quantum Algorithm for Linear Systems of Equations”. In: *Physical Review Letters* 103 (15 Oct. 2009), p. 150502.

[8] Vojtěch Havlíček et al. “Supervised learning with quantum-enhanced feature spaces”. In: *Nature* 567.7747 (Mar. 1, 2019), pp. 209–212.

[9] David B Hertz. “Risk Analysis in Capital Investment”. In: *Harvard Business Review* (Sept. 1979).

[10] Abhinav Kandala et al. “Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets”. In: *Nature* 549 (Sept. 13, 2017), pp. 242–247.

[11] Rachel Link. *How Much Caffeine Do Coke and Diet Coke Contain?* 2018. URL: https://www.healthline.com/nutrition/caffeine-in-coke.

[12] Ashley Montanaro. “Quantum speedup of Monte Carlo methods”. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 471.2181 (2015), p. 20150301.

[13] Michele Mosca and Marco Piani. *Quantum Threat Timeline*. 2019. URL: https://globalriskinstitute.org/publications/quantum-threat-timeline/.

[14] John Preskill. *Quantum Computing in the NISQ era and beyond*. URL: https://arxiv.org/abs/1801.00862.

[15] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126.

[16] Nikitas Stamatopoulos et al. *Option Pricing using Quantum Computers*. 2019. URL: https://arxiv.org/abs/1905.02666.

[17] Ewin Tang. *A quantum-inspired classical algorithm for recommendation systems*. 2019. URL: https://arxiv.org/pdf/1807.04271.pdf.

[18] Giacomo Torlai et al. *Precise measurement of quantum observables with neural-network estimators*. URL: https://arxiv.org/abs/1910.07596.

[19] P. Wittek. *Quantum Machine Learning. What quantum computing means to data mining*. Elsevier Science, 2016.

[20] Stefan Woerner and Daniel J. Egger. “Quantum risk analysis”. In: *npj Quantum Information* 5 (1 Feb. 8, 2019), pp. 198–201.

I

Foundations

2
They're Not Old, They're Classics

No simplicity of mind, no obscurity of station, can escape the universal duty of questioning all that we believe.

William Kingdon Clifford

When introducing quantum computing, it's easy to say "It's completely different from classical computing in every way!" Well that's fine, but to what exactly are you comparing it?

We start things off by looking at what a classical computer is and how it works to solve problems. This sets us up to later show how quantum computing replaces even the most basic classical operations with ones involving qubits, superposition, and entanglement.

Topics covered in this chapter

2.1 What's inside a computer?

If I were to buy a laptop today, I would need to think about the following kinds of hardware options:

- size and weight of the machine
- quality of the display
- processor and its speed
- memory and storage capacity

Three years ago I built a desktop gaming PC. I had to purchase and assemble and connect:

- the case
- power supply
- motherboard
- processor
- internal memory
- video card with a graphics processing unit (GPU) and memory
- internal hard drive and solid state storage
- internal Blu-ray drive
- wireless network USB device
- display
- speakers
- mouse and keyboard

As you can see, I had to make many choices. In the case of the laptop, you think about why you want the machine and what you want to do, and much less about the particular hardware. You don't have to make a choice about the manufacturers of the parts nor the standards that allow those parts to work together.

The same is true for smartphones. You decide on the mobile operating system, which then may decide the manufacturer, you pick a phone, and finally choose how much storage you want for apps, music, photos, and videos.

Of all the components above, I'm going to focus mainly on four of them: the processor, the "brain" for general computation; the GPU for specialized calculations; the memory, for holding information during a computation; and the storage, for long-term preservation of data used and produced by applications.

All these live on or are controlled by the motherboard and there is a lot of electronic circuitry that supports and connects them. My discussion about every possible independent or integrated component in the computer is therefore not complete.

Let's start with the storage. In applications like AI, it's common to process many megabytes or gigabytes of data to look for patterns and to perform tasks like classification. This information is held on either hard disk drives, introduced by IBM in 1956, or modern solid-state drives.

The smallest unit of information is a bit, and that can represent either the value 0 or the value 1.

The capacity of today's storage is usually measured in terabytes, which is $1,000 = 10^3$ gigabytes. A gigabyte is $1,000$ megabytes, which is $1,000$ kilobytes, which is $1,000$ bytes. Thus a terabyte is $10^{12} = 1,000,000,000$ bytes. A *byte* is 8 bits.

These are the base 10 versions of these quantities. It's not unusual to see a kilobyte being given as $1,024 = 2^{10}$ bytes, and so on. The values are slightly different, but close enough for most purposes.

The data can be anything you can think of – from music, to videos, to customer records, to presentations, to financial data, to weather history and forecasts, to the source for this book, for example. This information must be held reliably for as long as it is needed. For this reason this is sometimes called *persistent storage*. That is, once I put it on the drive I expect it to be there whenever I want to use it.

The drive may be within the processing computer or somewhere across the network. Data "on the cloud" is held on drives and accessed by processors there or pulled down to your machine for use.

What do I mean when I say that the information is reliably stored? At a high level I mean that it has backups somewhere else for redundancy. I can also insist that the data is encrypted so that only authorized people and processes have access. At a low level I want the data I store, the 0s and 1s, to always have exactly the values that were placed there.

Let's think about how plain characters are stored. Today we often use Unicode to represent over 100,000 international characters and symbols, including relatively new ones like emojis. [10] However, many applications still use the ASCII (also known as US-ASCII) character set, which represents a few dozen characters common in the United States. It uses 7 bits (0s or 1s) of a byte to correspond to these characters. Here are some examples:

Bits	Character	Bits	Character
0100001	!	0111111	?
0110000	0	1000001	A
0110001	1	1000010	B
0110010	2	1100001	a
0111100	<	1100010	b

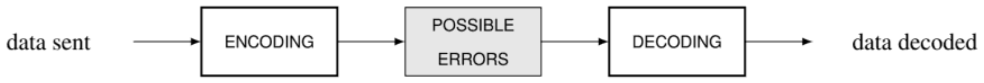
We number the bits from right to left, starting with 0:

$$\begin{array}{rccccccc} \text{character `a'}: & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ \text{position}: & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{array}$$

If something accidentally changes bit 5 in 'a' from 1 to 0, I end up with an 'A'. If this happened in text you might say that it wasn't so bad because it was still readable. Nevertheless, it is not the data with which we started. If I change bit 6 in 'a' to a 0, I get the character '1'. Data errors like this can change the spelling of text and the values of numeric quantities like temperature, money, and driver license numbers.

Errors may happen because of original or acquired defects in the hardware, extreme "noise" in the operating environment, or even the highly unlikely stray bit of cosmic radiation. Modern storage hardware is manufactured to very tight tolerances with extensive testing. Nevertheless, software within the storage devices can often detect errors and correct them. The goal of such software is to both detect and correct errors quickly using as little extra data as possible. This is called *fault tolerance*.

The idea is that we start with our initial data, encode it in some way with extra information that allows us to tell if errors occurred and perhaps how to fix them, do something with the data, then decode and correct the information if necessary.



One strategy for trying to avoid errors is to store the data many times. This is called a *repetition code*. Suppose I want to store an 'a'. I could save it five times:

1100001 **0**100001 **1**100001 **1**100001 **1**100001

The first and second copies are different in bit 6 and so an error occurred. Since four of the five copies agree, we might "correct" the error by deciding the actual value is 1100001. However, who's to say the other copies also don't have errors? There are more efficient ways of detecting and correcting errors than repetition, but it is a central concept that underlies several other schemes.

Another way to possibly detect an error is to use an *even parity bit*. We append one more bit to the data: if there is an odd number of 1 bits, then we precede the data with a 1. For an even number of 1s, we place a 0 at the beginning.

$$\begin{aligned} \mathsf{1100001} &\mapsto \mathbf{1}\mathsf{1100001} \\ \mathsf{1100101} &\mapsto \mathbf{0}\mathsf{1100101} \end{aligned}$$

If we get a piece of data and there are an odd number of 1s, we know at least one bit is wrong.

If errors continue to occur within a particular region of storage, then the controlling software can direct the hardware to avoid using it. There are three processes within in our systems that keep our data correct:

- *Error detection:* Discovering that an error has occurred.
- *Error correction:* Fixing an error with an understood degree of statistical confidence.
- *Error mitigation:* Preventing errors from happening in the first place through manufacturing or control.

To learn more

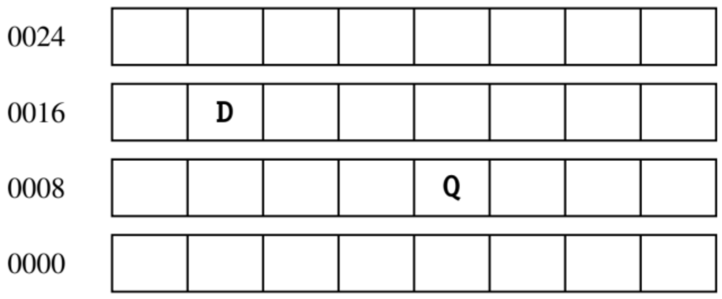
Many of the techniques and nomenclature of quantum error correction have their roots and analogs in classical use cases dating back to the 1940s. [2] [3] [8]

One of the roles of an operating system is to make persistent storage available to software through the handling of file systems. There are many schemes for file systems but you may only be aware that individual containers for data, the files, are grouped together into folders or directories. Most of the work regarding file systems is very low-level code that handles moving information stored in some device onto another device or into or out of an application.

From persistent storage, let's move on to the memory in your computer. I think a good phrase to use for this is "working memory" because it holds much of the information your system needs to use while processing. It stores part of the operating system, the running apps, and the working data the apps are using. Data or parts of the applications that are not needed immediately might be put on disk by a method called "paging." Information in memory can be gotten from disk, computed by a processor, or placed in memory directly in some other way.

The common rule is "more memory is good." That sounds trite, but low cost laptops often skimp on the amount or speed of the memory and so your apps run more slowly. Such laptops may have only 20% of the memory used by high-end desktop machines for video editing or games.

A memory location is accessed via an address:



The byte of data representing the character "**Q**" is at address 0012 while we have a "**D**" at address 0017. If you have a lot of memory, you need to use very large numbers as addresses.

Next up to consider is the central processing unit, the CPU, within a classical computer. It's a cliché, but it really is like the brain of the computer. It controls executing a sequence of instructions that can do arithmetic, move data in and out of memory, use designated extra-fast memory called *registers* within the processor, and conditionally jump somewhere else in the sequence. The latest processors can help in memory management for interpreted programming languages and can even generate random numbers.

Physically, CPUs are today made from transistors, capacitors, diodes, resistors, and the pathways that connect them all into an *integrated circuit*. We differentiate between these electronic circuits and the logic circuits that are implemented using them.

In 1965, Gordon Moore extrapolated from several years of classical processor development and hypothesized that we would be able to double the speed of and number of transistors on a chip roughly every two years. [6] When this started to lose steam, engineers worked out how to put multiple processing units within a computer. These are called *cores*.

Within a processor, there can be special units that perform particular functions. A *floating-point unit* (FPU) handles fast mathematical routines with numbers that contain decimals. An *arithmetic logic unit* (ALU) provides accelerated hardware support for integer arithmetic. A

CPU is not limited to having only one FPU or ALU. More may be included within the architecture to optimize the chip for applications like High Performance Computing (HPC).

Caches within a CPU improve performance by storing data and instructions that might be used soon. For example, instead of retrieving only a byte of information from storage, it's often better to pull in several hundred or thousand bytes near it into fast memory. This is based on the assumption that if the processor is now using some data it will soon use other data nearby. Very sophisticated schemes have been developed to keep the cores busy with all the data and instructions they need with minimal waiting time.

You may have heard about 32- or 64-bit computers and chosen an operating system based on one or the other. These numbers represent the *word size* of the processor. This is the "natural" size of the piece of data that the computer usually handles. It determines how large an integer the processor can handle for arithmetic or how big an address it can use to access memory.

In the first case, for a 32-bit word we use only the first 31 bits to hold the number and the last to hold the sign. Though there are variations on how the number is stored, a common scheme says that if that sign bit is 1 the number is negative, and zero or positive if the bit is 0.

For addressing memory, suppose as above that the first byte in memory has address 0. Given a 32-bit address size, the largest address is $2^{32} - 1$. This is a total of 4,294,967,296 addresses, which says that 4 gigabytes is the largest amount of memory with which the processor can work. With 64 bits you can "talk to" much more data.

Question 2.1.1

What is the largest memory address a 64-bit processor can access?

In advanced processors today, data in memory is not really retrieved or placed via a simple integer address pointing to a physical location. Rather, a *memory management unit* (MMU) translates the address you give it to a memory location somewhere within your computer via a scheme that maps to your particular hardware. This is called *virtual* memory.

In addition to the CPU, a computer may have a separate graphics processing unit (GPU) for high-speed calculations involving video, particularly games and applications like augmented and virtual reality. The GPU may be part of the motherboard or on a separate plug-in card with 2, 4, or more gigabytes of dedicated memory. These separate cards can use a lot of power and generate much heat but they can produce extraordinary graphics.

Because a GPU has a limited number of highly optimized functions compared to the more general purpose CPU, it can be much faster, sometimes hundreds of times quicker at certain operations. Those kinds of operations and the data they involve are not limited to graphics. Linear algebra and geometric-like procedures make GPUs good candidates for some AI algorithms. [11] Even cryptocurrency miners use GPUs to try to find wealth through computation.

A quantum computer today does not have its own storage, memory, FPU, ALU, GPU, or CPU. It is most similar to a GPU in that it has its own set of operations through which it may be able to execute some special algorithms significantly faster. How much faster? I don't mean twice as fast, I mean thousands of times faster.

A GPU is a variation on the classical architecture while a quantum computer is something entirely different.

You cannot take a piece of classical software or a classical algorithm and directly run it on a quantum system. Rather, quantum computers work together with classical ones to create new hybrid systems. The trick is understanding how to meld them together to do things that have been intractable to date.

2.2 The power of two

For a system based on 0s and 1s, the number 2 shows up a lot in classical computing. This is not surprising because we use binary arithmetic, which is a set of operations on base 2 numbers.

Most people use base 10 for their numbers. These are also called decimal numbers. We construct such numbers from the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, which we often call digits. Note that the largest digit, 9, is one less than 10, the base.

A number such as 247 is really shorthand for the longer $2 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$. For $1,003$ we expand to $1 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 3 \times 10^0$. In these expansions we write a sum of digits between 0 and 9 multiplied by powers of 10 in decreasing order with no intermediate powers omitted.

We do something similar for binary. A binary number is written as a sum of bits (0 or 1) multiplied by powers of 2 in decreasing order with no intermediate powers omitted. Here are some examples:

$$\begin{aligned}
0 &= 0 \times 2^0 \\
1 &= 1 \times 2^0 \\
10 &= 1 \times 2^1 + 0 \times 2^0 \\
1101 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0
\end{aligned}$$

The "10" is the binary number 10 and not the decimal number 10. Confirm for yourself from the above that the binary number 10 is another representation for the decimal number 2. If the context doesn't make it clear whether binary or decimal is being used, I use subscripts like 10_2 and 2_{10} where the first is base 2 and the second is base 10.

If I allow myself two bits, the only numbers I can write are 00, 01, 10, and 11. 11_2 is 3_{10} which is $2^2 - 1$. If you allow me 8 bits then the numbers go from 00000000 to 11111111. The latter is $2^8 - 1$.

For 64 bits the largest number I can write is a string of sixty four 1s which is

$$2^{64} - 1 = 18,446,744,073,709,551,615.$$

This is the largest positive integer a 64-bit processor can use.

We do binary addition by adding bits and carrying.

$$\begin{aligned} 0 + 0 &= 0 \\ 1 + 0 &= 1 \\ 0 + 1 &= 1 \\ 1 + 1 &= 0 \text{ carry } 1 \end{aligned}$$

Thus while $1 + 0 = 1$, $1 + 1 = 10$. Because of the carry we had to add another bit to the left. If we were doing this on hardware and the processor did not have the space to allow us to use that extra bit, we would be in an overflow situation. Hardware and software that do math need to check for such a condition.

2.3 True or false?

From arithmetic let's turn to basic logic. Here there are only two values: true and false. We want to know what kinds of things we can do with one or two of these values.

The most interesting thing you can do to a single logical value is to replace it with the other. Thus, the **not** operation turns true into false, and false into true:

not true = false

not false = true

For two inputs, which I call p and q, there are three primary operations **and**, **or**, and **xor**. Consider the statement "We will get ice cream only if you **and** your sister clean your rooms." The result is the truth or falsity of the statement "we will get ice cream."

If neither you nor your sister clean your rooms, or if only one of you clean your room, then the result is false. If both of you are tidy, the result is true, and you can start thinking about ice cream flavors and whether you want a cup or a cone.

Let's represent by (you, sister) the different combinations of you and your sister, respectively, having done what was asked. You will not get ice cream for (false, false), (true, false), or (false, true). The only combination that is acceptable is (true, true). We express this as

true **and** true = true

true **and** false = false

false **and** true = false

false **and** false = false

More succinctly, we can put this all in a table

p = you	q = your sister	p **and** q
true	true	true
true	false	false
false	true	false
false	false	false

where the first column has the values to the left of the **and**, and the second column has the values to the right of it. The rows are the values and the results. This is called a "truth table."

Another situation is where we are satisfied if at least one of the inputs is true. Consider "We will go to the movie if you **or** your sister feed the dog." The result is the truth or falsity of the statement "we will go to the movie."

p	q	p **or** q
true	true	true
true	false	true
false	true	true
false	false	false

Finally think about a situation where we care about one and only one of the inputs being true. This is similar to **or** except in the case (true, true) where the result is false. It is called an "exclusive or" and written **xor**. If I were to say, "I am now going to the restaurant or going to the library," then one of these can be true but not both, assuming I mean my very next destination.

Question 2.3.1

How would you state the inputs and result statement for this **xor** example?

p	q	p **xor** q
true	true	false
true	false	true
false	true	true
false	false	false

There are also versions of these that start with **n** to mean we apply **not** to the result. This means we apply an operation like **and**, **or**, or **xor** and then flip true to false or false to true. This "do something and then negate it" is not common in spoken or written languages but they are quite useful in computer languages.

nand is defined this way:

true **nand** false = **not** (true **and** false) = true

It has this table of values:

p	q	p **nand** q
true	true	false
true	false	true
false	true	true
false	false	true

I leave it to you to work out what happens for **nor** and **nxor**. These **n**-versions seem baroque and excessive now but several are used in the next section.

Question 2.3.2

Fill in the following tables based on the examples and discussion above.

p	*q*	*p* **nor** *q*
true	true	
true	false	
false	true	
false	false	

p	*q*	*p* **nxor** *q*
true	true	
true	false	
false	true	
false	false	

Instead of true and false we could have used 1 and 0, respectively, which hints at some connection between logic and arithmetic.

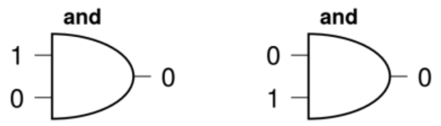
2.4 Logic circuits

Now that we have a sense of how the logic works, we can look at logic circuits. The most basic logic circuits look like binary relationships but more advanced ones implement operations for addition, multiplication, and many other mathematical operations. They also manipulate basic data. Logic circuits implement algorithms and ultimately the apps on your computer or device.

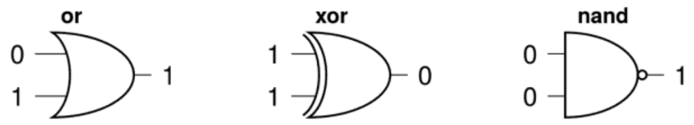
We begin with examples of the core operations, also called *gates*.

To me, the standard gate shapes used in the United States look like variations on spaceship designs.

Rather than use true and false, we use 1 and 0 as the values of the bits coming into and out of gates.

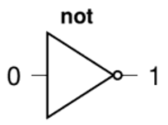


This gate has two inputs and one output. It is not reversible because it produces the same output with different inputs. Given the 0 output, we cannot know which example produced it. Here are the other gates we use, with example inputs:



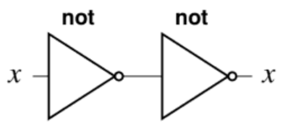
The symbol “⊕” is frequently used for the **xor** operation.

The **not** gate has one input and one output. It is reversible: if you apply it twice you get back what you started with.

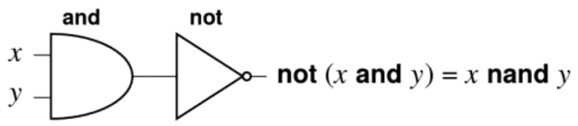


People who study electrical engineering see these gates and the logic circuits you can build from them early in their training. This is not a book about electrical engineering. Instead, I want you to think of the above as literal building blocks. We plug them together, pull them apart, and make new logic circuits. That is, we'll have fun with them while getting comfortable with the notion of creating logic circuits that do what we want.

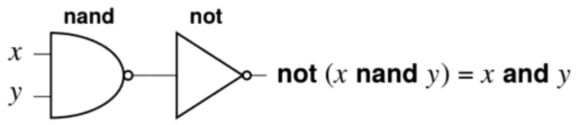
I connect the output of one **not** gate to the input of another to show you get the same value you put in. x can be 0 or 1.



If we didn't already have a special **nand** gate we could build it.

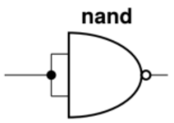


Note how we can compose gates to get other gates. We can build **and** from **nand** and **not**.



From this you can see that we could shrink the collection of gates we use if we desired. From this last example, it's technically redundant to have **and**, **nand**, and **not**, but it is convenient to have them all. Let's keep going to see what else we can construct.

Even though a gate like **nand** has two inputs, we can push the same value into each input. We show it this way:

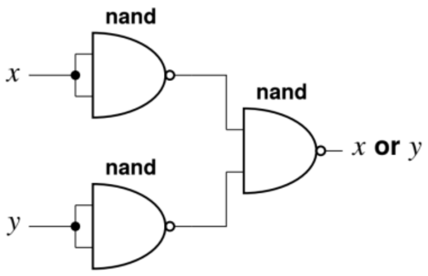


What do we get when we start with 0 or 1 and run it through this logic circuit? 0 **nand** 0 = 1 and 1 **nand** 1 = 0. This behaves exactly like **not**! This means if we really wanted to, we could get rid of **not**. With this, having only **nand** we could drop **not** and **and**. It's starting to seem that **nand** has some essential building block property.

By stringing together four **nand** gates, we can create an **xor** gate. It takes three to make an **or** gate. It takes one more of each to make **nxor** and **nor** gates, respectively.

Every logical gate can be constructed from a logic circuit of only **nand** gates. The same is true for **nor** gates. **nand** and **nor** gates are called *universal* gates because they have this property. [7]

It would be tedious and very inefficient to have all the basic gates replaced by multiple **nand** gates, but you can do it. This is how we would build **or**:



For the binary logic gates that have two inputs and one output, there are only eight possibilities, the four possible inputs 0/0, 0/1, 1/0, and 1/1 together with the outputs 0 and 1. Like **or** above, you can string together combinations of **nand**s to create any of these eight gates.

Question 2.4.1

Show how to create the **nor** gate only from **nand** gates.

We looked at these logic circuits to see how classical processing is done at a very low level. We return to circuits and the idea of universal gates again for quantum computing in section 9.2.

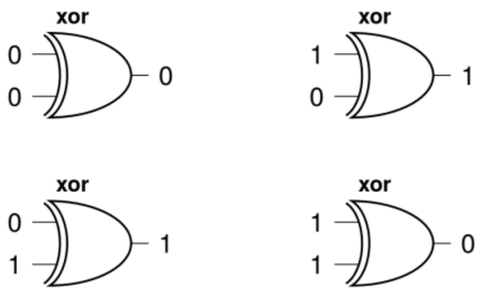
So far this has really been a study of classical gates for their own sakes. The behavior, compositions, and universality are interesting, but don't really do much fascinating yet. Let's do some math!

2.5 Addition, logically

Using binary arithmetic as we discussed in section 2.2,

$$\begin{aligned} 0 + 0 &= 0 \\ 1 + 0 &= 1 \\ 0 + 1 &= 1 \\ 1 + 1 &= 0 \text{ carry } 1 \end{aligned}$$

Focusing on the value after the equal signs and temporarily forgetting the carrying in the last case, this is the same as what **xor** does with two inputs.



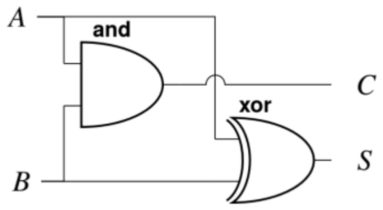
We did lose the carry bit but we limited ourself to having only one output bit. What gate operation would give us that 1 carry bit only if both inputs were also 1, and otherwise return 0? Correct, it's **and**! So if we can combine the **xor** and the **and** and give ourselves two bits of output, we can do simple addition of two bits.

Question 2.5.1

Try drawing a circuit that would do this before peeking at what follows. You are allowed to clone the value of a bit and send it to two different gates.

Question 2.5.2

Did you peek?

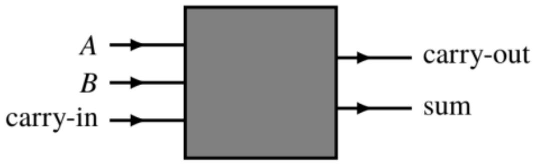


where A, B, S, and C are bits. The circuit takes two single input bits, A and B, and produces a 2-bit answer CD.

$$\begin{gathered} A + B = CS \\ 0 + 0 = 00 \\ 1 + 0 = 01 \\ 0 + 1 = 01 \\ 1 + 1 = 10 \end{gathered}$$

We call S the *sum* bit and C the *carry-out* bit. This circuit is called a half-adder since, as written, it cannot be used in the middle of a larger circuit. It's missing something. Can you guess what it is?

A full-adder has an additional input that is called the *carry-in*. This is the carry bit from an addition that might precede it in the overall circuit. If there is no previous addition, the carry-in bit is set to 0.



The square contains a circuit to handle the inputs and produce the two outputs.

Question 2.5.3

What does the circuit look like?

By extending this to more bits with additional gates, we can create classical processors that implement full addition. Other arithmetic operations like subtraction, multiplication, and division are implemented and often grouped within the *arithmetic logic unit* (ALU) of a classical processing unit.

For addition, the ALU takes multibit integer inputs and produces a multibit integer output sum. Other information can also be available from the ALU, such if the final bit addition caused an *overflow,* a carry-out that had no place to go.

ALUs contain circuits including hundreds or thousands of gates. A modern processor in a laptop or smartphone uses integers with 64 bits. Given the above simple circuit, try to estimate how many gates you would need to implement full addition.

Modern-day programmers and software engineers rarely deal directly with classical circuits themselves. Several layers are built on top of them so that coders can do what they need to do quickly.

If I am writing an app for a smartphone that involves drawing a circle, I don't need to know anything today about the low-level processes and circuits that do the arithmetic and cause the graphic to appear on the screen. I use a high-level routine that takes as input the location of the center of the circle, the radius, the color of the circle, and the fill color inside the circle.

At some point a person created that library implementing the high-level routine. Someone wrote the lower-level graphical operations. And someone wrote the very basic circuits that implement the primitive operations under the graphical ones.

Software is layered in increasing levels of abstraction. Programming languages like C, C++, Python, Java, and Swift hide the low-level details. Libraries for these languages provide reusable code that many people can use to piece together new apps.

There is always a bottom layer, though, and circuits live near there.

2.6 Algorithmically speaking

The word "algorithm" is often used generically to mean "something a computer does." Algorithms are employed in the financial markets to try to calculate the exact right moment and price at which to sell a stock or bond. They are used in artificial intelligence to find patterns in data to understand natural language, construct responses in human conversation, find manufacturing anomalies, detect financial fraud, and even to create new spice mixtures for cooking.

Informally, an algorithm is a recipe. Like a recipe for food, an algorithm states what inputs you need (water, flour, butter, eggs, etc.), the expected outcome (for example, bread), the sequence of steps you take, the subprocesses you should use (stir, knead, bake, cool), and what to do when a choice presents itself ("if the dough is too wet, add more flour").

We call each step an *operation* and give it a name as above: "stir," "bake," "cool," and so on. Not only do we want to overall process to be successful and efficient, but we construct the best possible operations for each action in the algorithm.

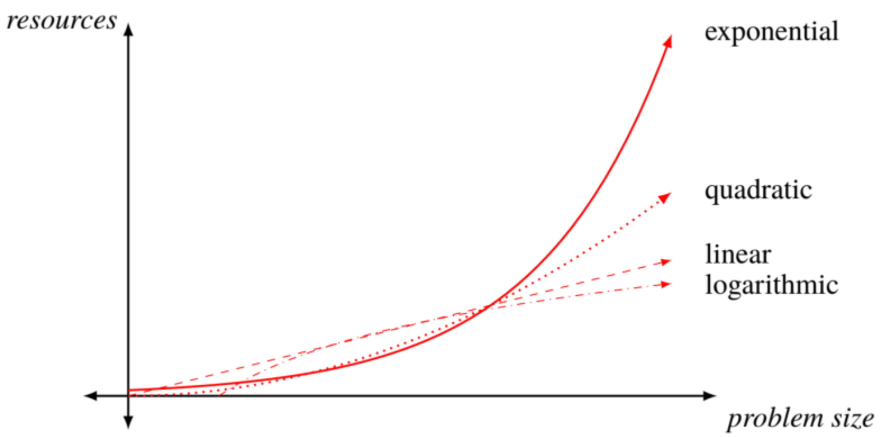
The recipe is not the real baking of the bread, you are doing the cooking. In the same way, an algorithm abstractly states what you should do with a computer. It is up to the circuits and higher-level routines built on circuits to implement and execute what the algorithm describes. There can be more than one algorithm the produces the same result.

Operations in computer algorithms might, for example, add two numbers, compare whether one is larger than another, switch two numbers, or store or retrieve a value from memory

Quantum computers do not use classical logical gates, operations, or simple bits. While quantum circuits and algorithms look and behave very differently from their classical counterparts, there is still the idea that we have data that is manipulated over a series of steps to get what we hope is a useful answer.

2.7 Growth, exponential and otherwise

Many people who use the phrase "exponential growth" use it incorrectly, somehow thinking it only means "very fast." Exponential growth involves, well, exponents. Here's a plot showing four kinds of growth: exponential, quadratic, linear, and logarithmic.



I've drawn them so they all intersect at a point but afterwards diverge. After the convergence, the logarithmic plot (dot dashed) grows slowly, the linear plot (dashed) continues as it did, the quadratic plot (dotted) continues upward as a parabola, and the exponential one shoots up rapidly.

Take a look at the change in the vertical axis, the one I've labeled *resources* with respect to the horizontal axis, labeled *problem size*. As the size of the problem increases, how fast does the amount of resources needed increase? Here a resource might be the time required for the algorithm, the amount of memory used during computation, or the megabytes of data storage necessary.

When we move a certain distance to the right horizontally for *problem size,* the logarithmic plot increases vertically at a rate proportional to the inverse of the size ($\frac{1}{\mathit{problem\ size}}$), the linear plot increases at a constant rate for *resources* that does not depend on *problem size*. The quadratic plot increases at a vertical rate that is proportional to *problem size*. The exponential plot increases at a rate that is proportional to its current *resources*.

The logarithm is only defined for positive numbers. The function $\log_{10}(x)$ answers the question "to what power must I raise 10 to get x?". When x is 10, the answer is 1. For x equals one million, the answer is 6. Another common logarithm function is $\log_2$ which substitutes 2 for 10 in these examples. Logarithmic functions grow *very* slowly.

Examples of growth are

$$\begin{aligned}
\mathit{resources} &= 2 \times \log_{10}(\mathit{problem\ size}) \\
\mathit{resources} &= 4 \times (\mathit{problem\ size}) \\
\mathit{resources} &= 0.3 \times (\mathit{problem\ size})^2 \\
\mathit{resources} &= 7.2 \times 3^{\mathit{problem\ size}}
\end{aligned}$$

for logarithmic, linear, quadratic, and exponential, respectively. Note the variable *problem size* in the exponent in the fourth case.

This means that for large problem size, the exponential plot goes up rapidly and then more rapidly and so on. Things can quickly get out of hand when you have this kind of positive exponential growth.

If you start with 100 units of currency and get 6% interest compounded once a year, you will have $100(1+0.06)$ after one year. After two you will have $100(1+0.06)(1+0.06) = 100(1.06)^2$. In general, after t years you will have $100(1.06)^t$. This is exponential growth. Your money will double in approximately 12 years.

Quantum computers will not be used over the next several decades to fully replace classical computers. Rather, quantum computing may help make certain solutions doable in a short time instead of being intractable. The power of a quantum computer potentially grows exponentially with the number of quantum bits, or qubits, in it. Can this be used to control similar growth in problems we are trying to solve?

2.8 How hard can that be?

Once you decide to do something, how long does it take you? How much money or other resources does it involve? How do you compare the worst way of doing it with the best?

When you try to accomplish tasks on a computer, all these questions come to bear. The point about money may not be obvious, but when you are running an application you need to pay for the processing, storage, and memory you use. This is true whether you paid to get a more powerful laptop or have ongoing cloud costs.

To end this chapter we look at classical complexity. To start, we consider sorting and searching and some algorithms for doing them.

Whenever I hear "sorting and searching" I get a musical ear worm for Bobby Lewis' 1960 classic rock song "Tossin' and Turnin'." Let me know if it is contagious.

2.8.1 Sorting

Sorting involves taking multiple items and putting them in some kind of order. Consider your book collection. You can rearrange them so that the books are on the shelves in ascending alphabetic order by title. Or you can move them around so that they are in descending order by the year of publication. If more than one book was published in the same year, order them alphabetically by the first author's last name.

When we ordered by title, that title was the *key* we looked at to decide where to place the book among the others. When we considered it by year and then author, the year was the *primary key* and the author name was the *secondary key*.

Before we sort, we need to decide how we compare the items. In words, we might say "is the first number less than the second?" or "is the first title alphabetically before the second?". The response is either true or false.

Something that often snags new programmers is comparing things that look like numbers either numerically or *lexicographically*. In the first case we think of the complete item as a number while in the second we compare character by character. Consider `54` and `8`. Numerically, the second is less than the first. Lexicographically, the first is less because the character `5` comes before `8`.

Therefore when coding you need to convert them to the same format. If you were to convert `-34,809` into a number, what would it be? In much of Europe the comma is used as the decimal point, while in the United States, the United Kingdom, and several other countries it is used to separate groups of three digits.

Now we look two ways of numerically sorting a list of numbers into ascending order. The first is called a "bubble sort" and it has several nice features, including simplicity. It is horribly inefficient, though, for large collections of objects that are not close to being in the right order.

We have two operations: *compare*, which takes two numbers and returns true if the first is less than the second, and false otherwise; and *swap*, which interchanges the two numbers in the list.

The idea of the bubble sort is to make repeated passes through the list, comparing adjacent numbers. If they are out of order, we swap them and continue through the list. We keep doing this until we make a full pass where no swaps were done. At that point the list is sorted. Pretty elegant and simple to describe!

Let's begin with a case where a list of four numbers is already sorted:

$$[-2, 0, 3, 7].$$

We compare -2 and 0. They are already in the correct order so we move on. 0 and 3 are also correct, so nothing to do. 3 and 7 are fine. We did three comparisons and no swaps. Because we did not have to interchange any numbers, we are done.

Now let's look at

$$[7, -2, 0, 3].$$

Comparing 7 and -2 we see the second is less than the first, so we swap them to get the new list

$$[-2, 7, 0, 3].$$

Next we look at 7 and 0. Again, we need to swap and get

$$[-2, 0, 7, 3].$$

Comparing 7 and 3 we have two more numbers that are out of order. Swapping them we get

$$[-2, 0, 3, 7].$$

So far we have done 3 comparisons and 3 swaps. Since the number of swaps is not zero, we pass through the list again. This time we do 3 more comparisons but no swaps and we are done. In total we did 6 comparisons and 3 swaps.

Now for the worst case with the list in reverse sorted order

$$[7, 3, 0, -2].$$

The first pass looks like

$[7, 3, 0, -2]$	swap first and second numbers
$[3, 7, 0, -2]$	swap second and third numbers
$[3, 0, 7, -2]$	swap third and fourth numbers
$[3, 0, -2, 7]$	

We did 3 comparisons and 3 swaps.

Second pass:

$[3, 0, -2, 7]$	swap first and second numbers
$[0, 3, -2, 7]$	swap second and third numbers
$[0, -2, 3, 7]$	compare third and fourth numbers but do nothing

We did 3 comparisons and 2 swaps.

Third pass:

$[0, -2, 3, 7]$	swap first and second numbers
$[-2, 0, 3, 7]$	compare second and third numbers but do nothing
$[-2, 0, 3, 7]$	compare third and fourth numbers but do nothing

We did 3 comparisons and 1 swap.

Fourth pass:

$[-2, 0, 3, 7]$	compare first and second numbers but do nothing
$[-2, 0, 3, 7]$	compare second and third numbers but do nothing
$[-2, 0, 3, 7]$	compare third and fourth numbers but do nothing

No swap, so we are done, and the usual 3 comparisons.

For this list of 4 numbers we did 12 comparisons and 6 swaps in 4 passes. Can we put some formulas behind these numbers for the worst case?

We had 4 numbers in completely the wrong order and so it took 4 full passes to sort them. For a list of length n we must do $n - 1$ comparisons, which is 3 in this example.

The number of swaps is the interesting number. On the first pass we did 3, on the second we did 2, the third we did 1, and the fourth we did none. So the number of swaps is

$$3 + 2 + 1 = (n - 1) + (n - 2) + \cdots + 1$$

where n is the length of the list. Obviously there is a pattern here.

There is a formula that can help us with this last sum. If we want to add up 1, 2, and so on through a positive integer m, we compute $\frac{m(m+1)}{2}$.

Question 2.8.1

Try this out for $m = 1$, $m = 2$, and $m = 3$. Now see if the formula still holds if we add in $m + 1$. That is, can you rewrite

$$\frac{m(m+1)}{2} + m + 1$$

as

$$\frac{(m+1)(m+2)}{2} \quad ?$$

If so, you have proved the formula by *induction*.

In our case, we have $m = n - 1$ and so we do a total of $\frac{(n-1)n}{2}$ swaps. For $n = 4$ numbers in the list, this is none other than the 6 swaps we counted by hand.

It may not seem so bad that we had to do 6 swaps for 4 numbers in the worst case, but what if we had 1,000 numbers in completely reversed order? Then the number of swaps would be

$$\frac{999 \times 1000}{2} = 499500$$

That's almost half a million swaps for a list of 1,000 numbers.

For 1 million numbers in the worst case it would be

$$\frac{999999 \times 1000000}{2} = 499999500000$$

which is 499 billion 999 million 500 thousand swaps. This is terrible. Rewriting

$$\frac{(n-1)n}{2} = \frac{n^2 - n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

we can see that the number of swaps grows with the *square* of the number of entries. In fact,

$$\text{number of swaps } \leq \frac{1}{2}n^2$$

for all $n \geq 1$. When we have this kind of situation, we say that our algorithm is $O\left(n^2\right)$, pronounced "big Oh of n squared."

More formally, we say that the number of interesting operations used to solve a problem on n objects is $O(f(n))$ if there is a positive real number c and an integer m such that

$$\text{number of operations } \leq c f(n)$$

once $n \geq m$, for some function f.

In our case, $c = \frac{1}{2}$, $f(n) = n^2$, and $m = 1$.

To learn more

In the area of computer science called *complexity theory*, researchers try to determine the best possible f, c, and m to match the growth behavior as closely as possible. [1, Chapter 3] [9, Section 1.4]

If an algorithm is $O(n^t)$ for some positive fixed number t then the algorithm is *of polynomial time*. Easy examples are algorithms that run in $O(n)$, $O\left(n^2\right)$, and $O\left(n^3\right)$ time.

If the exponent t is very large then we might have a very inefficient and impractical algorithm. Being of polynomial time really means that it is bounded above by something that is $O(n^t)$. We therefore also say that an $O(\log(n))$ algorithm is of polynomial time since it runs faster that an $O(n)$ algorithm.

Getting back to sorting, we are looking at the worst case for this algorithm. In the best case we do no swaps. Therefore when looking at a process we should examine the best case, the worst case, as well as the average case. For the bubble sort, the best case is $O(n)$ while the average and worst cases are both $O(n^2)$.

If you are coding up and testing an algorithm and it seems to take a very long time to run, you likely either have a bug in your software or you have stumbled onto something close to a worst case scenario.

Can we sort more efficiently than $O(n^2)$ time? How much better can we do?

By examination we could have optimized the bubble sort algorithm slightly. Look at the last entry in the list after each pass and think about how we could have reduced the number of comparisons. However, the execution time is dominated by the number of swaps, not the number of comparisons. Rather than tinkering with this kind of sort, let's examine another algorithm that takes a very different approach.

There are many sorting algorithms and you might have fun searching the web to learn more about the main kinds of approaches. You'll also find interesting visualizations about how the objects move around during the sort. Computer science students often examine the different versions when they learn about algorithms and data structures.

John von Neumann in the 1940s.
Photo subject to use via the Los Alamos National Laboratory notice.

The second sort algorithm we look at is called a *merge sort* and it dates back to 1945. It was discovered by John von Neumann. [4]

To better see how the merge sort works, let's use a larger data set with 8 objects and we use names instead of numbers. Here is the initial list:

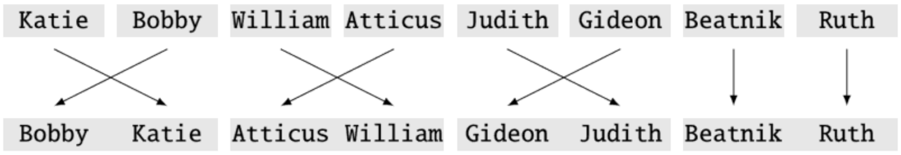
```
Katie   Bobby  William Atticus  Judith  Gideon  Beatnik  Ruth
```

We sort this in ascending alphabetic order. This list is now neither sorted nor in reverse order, so this is an average case.

We begin by breaking the list into 8 groups (because we have 8 names), each containing only one item.

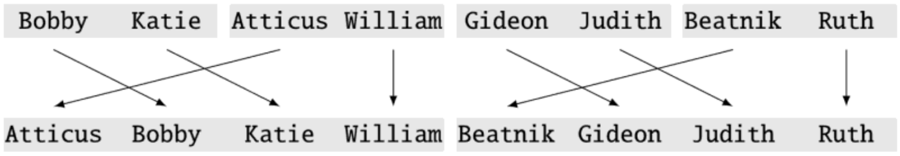
```
Katie  Bobby  William  Atticus  Judith  Gideon  Beatnik  Ruth
```

Trivially, each group is sorted within itself because there is only one name in it. Next, working from left to right pairwise, we create groups of two names where we put the names in the correct order as we merge.

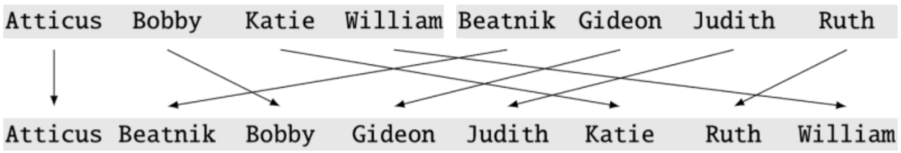


Now we combine the groups of two into groups of four, again working from left to right. We know the names are sorted within the group. Begin with the first name in the first pair. If it is less than the first name in the second pair, put it at the beginning of the group of four. If not, put the first name in the pair there.

Continue in this way. If a pair becomes empty, put all the names in the other pair at the end of the group of four, in order.



We finally create one group, merging in the names as we encounter them from left to right.



Among the variations of merge sort, this is called a *bottom-up* implementation because we completely break the data into chunks of size 1 and then combine.

For this algorithm we are not interested in swaps because we are forming new collections rather than manipulating an existing one. That is, we need to place a name in a new group no matter what. Instead the metric we care about is the number of comparisons. The analysis is non-trivial, and is available in algorithms books and on the web. The complexity of a merge sort is $O(n \log(n))$. [1]

This is a big improvement over $O(n^2)$. Forgetting about the constant in the definition of $O(\ )$ and using logarithms to base 10, for $n = 1000000 = 1$ million, we have $n^2 = 1000000000000 = 10^{12} = 1$ trillion, while $\log_{10}(1000000) \times 1000000 = 6000000 = 6 \times 10^6 = 6$ million. Would you rather do a million times more comparisons than there are names or 6 times the number of names?

In both bubble and merge sorts we end up with the same answer, but the algorithms used and their performance differ dramatically. Choice of algorithm is an important decision.

For the bubble sort we had to use only enough memory to hold the original list and then we moved around numbers within this list. For the merge sort in this implementation, we need the memory for the initial list of names and then we needed that much memory again when we moved to groups of 2. However, after that point we could reuse the memory for the initial list to hold the groups of 4 in the next pass.

By reusing memory repeatedly we can get away with using twice the initial memory to complete the sort. By being extra clever, which I leave to you and your research, this memory requirement can be further reduced.

Question 2.8.2

I just referred to storing the data in memory. What would you do if there were so many things to be sorted that the information could not all fit in memory? You would need to use persistent storage like a hard drive, but how?

Though I have focused on how many operations such as swaps and comparisons are needed for an algorithm, you can also look at how much memory is necessary and do a $O(\ )$ analysis of that. Memory-wise, bubble sort is $O(1)$ and merge sort is $O(n)$.

2.8.2 Searching

Here is our motivating problem: I have a collection S of n objects and I want to find out if a particular object *target* is in S. Here are some examples:

- Somewhere in my closet I think I have a navy blue sweater. Is it there and where? Here *target* = "my navy blue sweater."
- If I keep my socks sorted by the names of their predominant colors in my dresser drawer, are my blue argyle socks clean and available?
- I have a database of 650 volunteers for my charity. How many live in my town?
- I brought my kids to a magic show and I got volunteered to go up on stage. Where is the Queen of Hearts in the deck of cards the magician is holding?

With only a little thought, we can see that searching is at worst $O(n)$ unless you are doing something strange and questionable. Look at the first object. Is it *target*? If so, look at the second and compare. Continue if necessary until we get to the nth object. Either it is *target* or *target* is not in S. This is a *linear search* because we go in a straight line from the beginning to the end of the collection.

If *target* is in S, in the best case we find it the first time and in the worst it is the nth time. On average it takes $\frac{n}{2}$ attempts.

To do better than this classically we have to know more information:

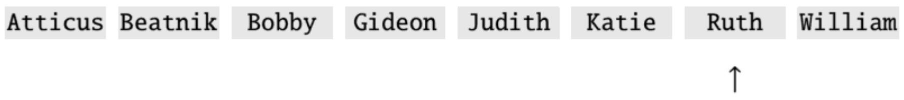
- Is S sorted?
- Can I access any object directly as in "give me the fourth object"? This is called *random access*.
- Is S simply a linear collection of objects or does it have a more sophisticated data structure?

If S only has one entry, look at it. If *target* is there, we win.

If S is a sorted collection with random access, I can do a *binary search*.

Since the word "binary" is involved, this has something to do with the number 2.

I now show this with our previously sorted list of names. The problem we pose is seeing if *target* = `Ruth` is in S. There are 8 names in the list. If we do a linear search it would take 7 tries to find `Ruth`.



Let's take a different approach. Let $m = \frac{n}{2} = 4$ which is near the midpoint of the list of 8 names.

In case m is not a whole number, we round up. Examine the mth = fourth name in S. That name is `Gideon`. Since S is sorted and `Gideon` < `Ruth`, `Ruth` cannot be in the first half of the list. With this simple calculation we have already eliminated half the names in S. Now we need only consider

`Judith` `Katie` `Ruth` `William`

There are 4 names and so we divide this in half to get 2. The second name is `Katie`. Since `Katie` < `Ruth`, `Ruth` is again not in the first half of this list. We repeat with the second half.

`Ruth` `William`

The list has length 2 and we divide this in half and look at the first name. `Ruth`! Where have you been?

We found `Ruth` with only 3 searches. If by any chance that last split and compare had not found `Ruth`, the remaining sublist would have had only one entry and that had to be `Ruth` since we assumed that name was in S. We located our target with only $3 = \log_2(8)$ steps.

Binary search is $O(log(n))$ in the worst case, but remember than we imposed the conditions that S was sorted and had random access. As with sorting, there are many searching techniques and data structures that can make finding objects quite efficient.

As an example of a data structure, look at the binary tree of our example names as shown in Figure 2.1. The dashed lines show our root to `Ruth`. To implement a binary tree in a computer requires a lot more attention to memory layout and bookkeeping.

Question 2.8.3

I want to add two more names to this binary tree, `Richard` and `Kristin`. How would you insert the names and rearrange the tree? What about if I delete `Ruth` or `Gideon` from the original tree?

Question 2.8.4

For extra credit, look up how *hashing* works. Think about the combined performance of searching and what you must do to keep the underlying data structure of objects in a useful form.

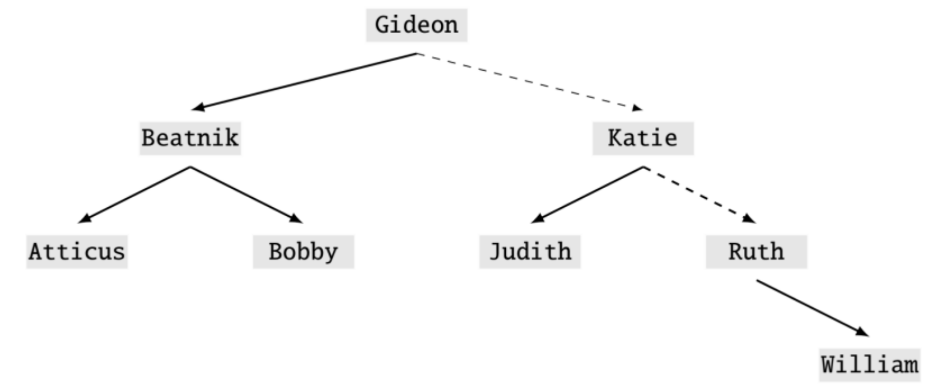


Figure 2.1: Binary tree

Entire books have been written on the related topics of sorting and searching. We return to this topic when we examine Grover's quantum algorithm for locating an item in an unsorted list without random access in only $O(\sqrt{n})$ time in section 9.7.

If we replace an $O(f(n))$ algorithm with an $O\left(\sqrt{f(n)}\right)$ one, we have made a *quadratic* improvement. If we replace it with a $O(\log(f(n)))$ algorithm, we have made an *exponential* improvement.

Suppose I have an algorithm that takes 1 million = 10^6 days to complete. That's almost 2,740 years! Forgetting about the constant in the $O()$ notation and using $\log_{10}$, a quadratic improvement would complete in $1,000 = 10^3$ days, which is about 2.74 years. An exponential improvement would give us a completion time of just 6 days.

To learn more

There are many kinds of sorting and searching algorithms and, indeed, algorithms for hundreds of computational use cases. Deciding which algorithm to use in which situation is very important to performance, as we have just seen. For some applications, there are algorithms to choose the algorithm to use! [5] [9]

2.9 Summary

Classical computers have been around since the 1940s and are based on using bits, 0s and 1s, to store and manipulate information. This is naturally connected to logic as we can think of a 1 or 0 as true or false, respectively, and vice versa. From logical operators like **and** we created real circuits that can perform higher-level operations like addition. Circuits implement portions of algorithms.

Since all algorithms to accomplish a goal are not equal, we saw that having some idea of measuring the time and memory complexity of what we are doing is important. By understanding the classical case we'll later be able to show where we can get a quantum improvement.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009.

[2] R.W. Hamming. "Error Detecting and Error Correcting Codes". In: *Bell System Technical Journal* 29.2 (1950).

[3] R. Hill. *A First Course in Coding Theory*. Oxford Applied Linguistics. Clarendon Press, 1986.

[4] Institute for Advanced Study. *John von Neumann: Life, Work, and Legacy*. URL: https://www.ias.edu/von-neumann.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd ed. Addison Wesley Longman Publishing Co., Inc., 1998.

[6] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965), p. 114.

[7] Ashok Muthukrishnan. *Classical and Quantum Logic Gates: An Introduction to Quantum Computing*. URL: http://www2.optics.rochester.edu/users/stroud/presentations/muthukrishnan991/LogicGates.pdf.

[8] Oliver Pretzel. *Error-correcting Codes and Finite Fields*. Student Edition. Oxford University Press, Inc., 1996.

[9] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley Professional, 2011.

[10] The Unicode Consortium. *About the Unicode® Standard*. URL: http://www.unicode.org/standard/standard.html.

[11] B. Tuomanen. *Explore high-performance parallel computing with CUDA*. Packt Publishing, 2018.

3

More Numbers than You Can Imagine

The methods of theoretical physics should be applicable to all those branches of thought in which the essential features are expressible with numbers.

Paul Dirac
1933 Nobel Prize Banquet Speech

People use numbers for counting, percentages, ratios, prices, math homework, their taxes, and other practical applications.

$$
\begin{array}{cccc}
1 & 0 & -1 & 9.99999 \\
-\sqrt{2}+1 & \dfrac{22}{7} & 3.14159265\ldots & \pi
\end{array}
$$

All these are examples of real numbers. In this chapter we look at the properties of real numbers and especially those of subsets like the integers. We extend those to other collections like the complex numbers that are core to understanding quantum computing.

For example, a quantum bit, or qubit, is defined as a pair of complex numbers with additional properties. Here we begin to lay the foundation for the algebraic side of quantum computing. In the next chapter we turn to geometry.

Topics covered in this chapter

3.1 Natural numbers

While there are special and famous numbers like π, the numbers we use for counting are much simpler: 1, 2, 3, ... I might say “Look, there is 1 puppy, 2 kittens, 3 cars, and 4 apples.” If you give me 2 more apples, I will have 6. If I give my sister 1 of them, I will have 5. If I buy 2 more bags of 5 apples, I will have 15 in total, which is 3×5.”

The set of natural numbers is the collection of increasing values

$$\{1, 2, 3, 4, 5, 6, 7, \dots\}$$

where we get from one number to the next by adding 1. 0 is not included. The braces "{" and "}" indicate we are talking about the entire set of these numbers.

When we want to refer to some arbitrary natural number but not any one specifically, we use a variable name like n and m.

The set of natural numbers is infinite. Suppose otherwise and that some specific number n is the largest natural number. But then $n + 1$ is larger and is a natural number, by definition. This *proof by contradiction* shows that the original premise, that there is a largest natural number, is false. Hence the set is infinite.

To avoid writing out "natural numbers" repeatedly, I sometimes abbreviate the collection of all natural numbers by using $\mathbb{N}$. What can we do if we restrict ourselves to working only with $\mathbb{N}$?

First, we can add them via the familiar arithmetic rules. $1 + 1 = 2$, $3 + 4 = 7$, $999998 + 2 = 1000000$, and so on.

Addition is so key to natural numbers that we consider it an essential part of the definition. In fact, we did: we described the values in the set, $\{1, 2, \dots\}$, but then also pointed out that we get from one value to the next by adding 1.

I've started with these basic numbers to make this point clear: we're not only concerned with a number here or a number there. We want to think about the entire collection and what we can do with them via operations like addition.

If we have two natural numbers and we add them together using "+", we always get another natural number. Hence $\mathbb{N}$ is closed under addition. The idea of closure or being closed with respect to doing something means that after we do it, the result is in the collection.

To choose something more exotic than basic arithmetic, consider the square root operation. The square root of 1 is still 1 and so a natural number, as is the square root of 4. But $\sqrt{2}$ is not a natural number and is, in fact, an irrational number.

$$\sqrt{2} = 1.41421356237\dots$$

$\mathbb{N}$ is not closed under the square root operation.

In $\mathbb{N}$ addition is commutative: $4 + 11 = 11 + 4$ and $n + m = m + n$ in general. No matter which order you count things, you always get the same answer.

What about subtraction, which is in some sense the complement of addition? Since $3+4=7$ then $3=7-4$ and $4=7-3$. We can also state the last as "4 is 7 minus 3, or 7 take away 3, or 7 subtract 3."

For all natural numbers n and m we always have $n+m$ in $\mathbb{N}$. However, we have $n-m$ in $\mathbb{N}$ only if $n>m$. $24-17=7$ is a natural number but $6-6$ is not a natural number because 0 is not in $\mathbb{N}$ by definition. $17-24$ is not a natural number since the smallest natural number is 1. $\mathbb{N}$ is not closed under subtraction.

We can use comparisons like "<" and ">" to tell if one natural number is less than or greater than another, respectively, in addition to testing for equality with "=". Because we can compare any two numbers in $\mathbb{N}$ in this way, we say that the natural numbers are ordered. Since we have comparison operations, we can sort collections of natural numbers in ascending or descending ways.

Given that we have addition, we can define multiplication "×" by saying that $n\times m$ is m added to itself n times. In particular, $1\times n=n\times 1=n$. Multiplication distributes over addition: $3\times(8+11)=(3\times 8)+(3\times 11)=57$.

Multiplication is commutative like addition: $n\times m=m\times n$. It's just a question of how you group things for counting:

$$\begin{aligned}
3\times 7 &= 7+7+7\\
&= (3+3+1)+(2+3+2)+(1+3+3)\\
&= 3+3+(1+2)+3+(2+1)+3+3\\
&= 3+3+3+3+3+3+3\\
&= 7\times 3.
\end{aligned}$$

$\mathbb{N}$ is closed under multiplication but is not closed under division. $1/3$ is not a natural number, for example, even if $4/2$ is.

For natural numbers, the definition of multiplication follows directly from addition. For more sophisticated mathematical collections, multiplication can be much more complicated.

Let's start extending the collection to eliminate some of these problems regarding closure.

3.2 Whole numbers

If we append 0 to $\mathbb{N}$ as a new smallest value we get the whole numbers, denoted $\mathbb{W}$. The whole numbers are not used a lot by themselves in mathematics but let's see what we get with this additional value.

We are still closed under addition and multiplication and not closed under division. We do now have to watch out for division by zero. Expressions like $3 - 3$ or $n - n$ in general are in $\mathbb{W}$, so that's a little better for subtraction but this does not give us closure.

So far, there's not much that we've gained, it seems. Or have we?

0 is an *identity element* for addition, which is a new concept for us to consider. I've put it in bold to show how special it is. This is a unique (meaning there is one and only one) number such that for any whole number w we have $w + \mathbf{0} = \mathbf{0} + w = w$.

Thus $14 + \mathbf{0} = \mathbf{0} + 14 = 14$. Also, $\mathbf{0} \times w = w \times \mathbf{0} = \mathbf{0}$.

For the whole numbers $\mathbb{W}$ we have a collection of values

$$\{\mathbf{0}, 1, 2, 3, \ldots\},$$

an operation "+", and an identity element **0** for "+".

You may have realized by now that when we discussed the natural numbers we could have also noted that **1** is an identity element for multiplication. So let's restate everything we know about $\mathbb{N}$ and $\mathbb{W}$:

The set of natural numbers $\mathbb{N}$ is the infinite ordered collection of values $\{\mathbf{1}, 2, 3, 4, \ldots\}$ with a commutative operation "+" called *addition*. We get from one natural number to the next larger by adding **1**. $\mathbb{N}$ is closed under addition. We also have a commutative operation *multiplication* "$\times$" with identity element **1**. Multiplication distributes over addition. $\mathbb{N}$ is closed under multiplication. $\mathbb{N}$ is not closed under subtraction or division defined in the usual manner.

The set of whole numbers $\mathbb{W}$ is the infinite ordered extension of $\mathbb{N}$ formed by adding a new smallest value **0**. **0** is an identity element for addition. $\mathbb{W}$ is closed under the commutative addition and multiplication operations but not under subtraction or division.

I hope you agree that we've come a long way from only considering the numbers 1, 2, 3, 4, 5, ... We've gone from thinking about counting and specific values to entire sets of numbers and their properties. Though I'll stop putting them in bold, 0 and 1 are not any random numbers; they play very special roles. Later on we'll see other "**0**-like" and "**1**-like" objects that are more than simple numbers.

Since we have addition and multiplication, we can define exponentiation. If a and w are whole numbers, w^a equals w multiplied by itself a times. Note the similarity with how we defined multiplication from addition:

$$3^7 = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3.$$

This means that $w^1 = w$. Less intuitively at the moment, $w^0 = 1$ even when $w = 0$.

What do we get when we multiply two expressions with exponents? When the base is the same (the thing we are raising to a power), we add the exponents:

$$\begin{aligned} 2^3 \times 2^4 &= (2 \times 2 \times 2) \times (2 \times 2 \times 2 \times 2) \\ &= 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \\ &= 2^7 = 2^{3+4}. \end{aligned}$$

In general, $w^a \times w^b = w^{a+b}$. Also,

$$w^a \times w^0 = w^{a+0} = w^a = w^a \times 1$$

further showing why $w^0 = 1$ makes sense.

We omit the "×" when it is clear that the context is multiplication: $2^3 \times 2^4 = 2^3 2^4 = 2^7$.

3.3 Integers

People are sometimes confused by negative numbers when they first encounter them. How can I have a negative amount of anything? I can't physically have fewer than no apples, can I?

To get around this, we introduce the idea that a positive number of things or amount of money means something that you yourself *have*. A negative number or amount means what you *owe* someone else.

If you have \$100 or €100 or ¥100 and you write a check or pay a bill electronically for 120, one of two things will likely happen. The first option is for the payment to fail and your bank may charge you a fee. The second is that the bank will pay the full amount, let you know that you are overdrawn, and charge you a fee. You will then need to pay the amount overdrawn quickly or have it paid from some other account.

Whatever currency you used, you started with 100 and ended up with -20 before repayment. You owed the bank 20. If you deposit 200 in your account immediately, your balance will be 180, which is $-20 + 200$.

The *integers*, denoted $\mathbb{Z}$, take care of the problem of the whole numbers not being closed under subtraction. We define an operation "–", called *negation*, for each whole number n such that $-0 = 0$ and $-n$ is a new value such that $n + -n = 0$. This new extended set of values with the operations and properties we will discuss is called the *integers*.

Integers like 1, 12, and 345 are said to be *positive*. More precisely, any integer that is also a natural number or a whole number greater that 0 is positive. Positive integers are greater than 0.

Integers like -4, -89, and -867253 are *negative*. Said differently, for n a natural number, any integer of the form $-n$ is negative. Negative integers are less than 0. 0 is neither positive nor negative.

Negation has the property that $--n = n$. Negation reverses the order of the values: since $4 < 7$ then $-4 > -7$, which is the same as $-7 < -4$. The set of ordered values in the integers looks like

$$\{\ldots, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots\}.$$

Negative signs cancel out in multiplication and division: $-1 \times -1 = 1$ and $-1/-1 = 1$. For any integers n and m we have relationships like $n \times -m = -n \times m = -(n \times m)$.

If n is a whole number then $(-1)^n$ is 1 if n is even and is -1 if n is odd.

Given an integer n, we define the *absolute value* of n to be 0 if n is 0, n if n is positive, and $-n$ if n is negative. We use vertical bars $|n|$ to denote the absolute value of n. Therefore

$$|n| = \begin{cases} n & \text{when } n > 0 \\ 0 & \text{when } n = 0 \\ -n & \text{when } n < 0. \end{cases}$$

Here are some examples:

$$\begin{aligned} |-87| &= 87 \\ |0| &= 0 \\ |231| &= 231 \end{aligned}$$

Informally, you get the absolute value by discarding the negative sign in front of an integer in case it is present. We could also say that the absolute value of an integer is how far it is away from 0 where we don't worry whether it is less than or greater than 0.

For integers n and m, we always have $|nm| = |n| \times |m|$ and $|n + m| \leq |n| + |m|$.

Absolute value is a measurement of size or length and it generalizes to other concepts in algebra and geometry. In fact, for a qubit, the absolute value is related to the probability of getting one answer or another in a computation.

Examples

$n + 3$ gives us n increased by 3 and $n + -3 = n - 3$ yields n decreased by 3.

- $7 + 3$ means we increase 7 by 3 to get 10.
- $7 + -3$ means we decrease 7 by 3 to get 4.
- $-7 + 3$ means we increase -7 by 3 to get -4.
- $-7 + -3$ means we decrease -7 by 3 to get -10.

Considering the usual rules for addition and the above, $n + -m = n - m$ and $n - -m = n + m$ for any two integers n and m.

With this you can see that the integers are closed under subtraction: if you subtract one integer from another you get another integer. Given addition and negation we can get away without subtraction, but we keep it for convenience and to reduce the complexity of expressions.

You may have been first introduced to these rules and properties when you were a preteen. I repeated them here so that you would think more generally about the operations of addition, subtraction, and negation as they applied to arbitrary integers versus simply doing some arithmetic.

Returning to negation, for any integer there is one and only one number you can add to it and get 0. If we have 33, we would add -33 to it to get 0. If we started with -74 then adding 74 yields 0. 0 is significant here because it is the identity element for addition.

Any number with absolute value 1 is called a *unit*. The integers have two: 1 and -1.

A *prime* is a positive integer greater than 1 whose only factors under multiplication are 1 and itself. These are primes:

$$2 = 1 \times 2 \qquad 3 = 1 \times 3 \qquad 37 = 1 \times 37$$

These are not primes:

$$\begin{array}{ccc} 0 & 1 & -25 = -5 \times 5 \\ 4 = 2 \times 2 & 12 = 3 \times 4 = 2^2 \times 3 & 500 = 2^2 \times 5^3 \end{array}$$

When one number divides another evenly we use "|" between them. Since 5 divides 500, we write $5|500$. An integer that is the product of two or more primes (which may be the same, such as 7×7) is composite.

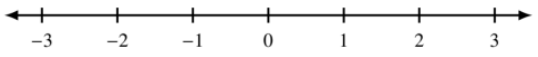
3×2 and 2×3 are equivalent factorizations of 6. We usually display factorizations with the individual factors going from small to large.

You can uniquely factor a non-zero integer into zero or more primes times a unit, with some of the primes possibly repeated. In the factorization of 500 above, the prime 2 was repeated twice and the prime 5 was repeated three times.

There are an infinite number of primes. The study of primes and their generalizations is relevant to several areas of mathematics, especially number theory.

The integers $\mathbb{Z}$ is the infinite ordered extension of $\mathbb{W}$ formed by appending the negative values $-1, -2, -3, -4, \ldots$. The integers have a unique identity element 0 such that for any integer n there exists a unique additive inverse $-n$ such that $n + -n = 0$. $\mathbb{Z}$ is closed under the commutative addition and multiplication operations, and subtraction, but not under division. Multiplication distributes over addition.

Taking a more geometric approach to the integers, we can draw the familiar number line with the negative values to the left of 0 and positive values to the right.



This is a visualization to help you think about the integers. The number line is *not* the set of integers but connects the algebra with a geometric aid.

Negating an integer corresponds to reflecting it to the other side of 0. Negating an integer twice means moving it across 0 and then back again to where it started. Hence double negation effectively does nothing. The absolute value means measuring how far to the left or right an integer is from 0.

Adding 0 does not move the position of a number on the line. Adding a positive integer means moving it that many units to the right, as does subtracting a negative integer. Adding a negative integer means moving it the absolute value of the integer units to the left, as does subtracting a positive integer.

Question 3.3.1

Using the number line, think about why negation reverses the order of two integers.

A line is one-dimensional. It takes only one number to exactly position a point anywhere on the line. Hence we can write (7) as the coordinate of the point exactly 7 units to the right of 0. With respect to this line, we call 0 the *origin*. Here is yet another special role for 0.

Mathematicians often work by taking a problem in one domain and translating it into another that they understand better or have better tools and techniques. Here I've shown some of the ways you can translate back and forth between the algebra and geometry of integers.

3.4 Rational numbers

The rational numbers, denoted $\mathbb{Q}$, take care of the problem of the integers not being closed under division by non-zero values.

3.4.1 Fractions

Let's start by talking about fractions, also known as the rational numbers, the way you may have been first introduced to them. This is elementary but useful to review to relate to what we have in the big picture with $\mathbb{Q}$.

Given a loaf of bread, if we cut it right down the middle, we say we have divided it into halves. Fraction-wise, one-half = $1/2$. The two halves equal one whole loaf, so $1/2+1/2 = 2\times 1/2 = 1$. Two halves is $2/2$, which is 1. Four halves would make two loaves: $4/2 = 2$.

Considering whole loaves, $1/1$ is one loaf, $2/1$ is two loaves, and $147/1$ is one hundred and forty-seven loaves. We can represent any integer n as a fraction $n/1 = \frac{n}{1}$.

To multiply fractions, we multiply the tops (*numerators*) together and put those over the product (the result of multiplication) of the bottoms (*denominators*) and then simplify the result, a process I discuss below.

If we get another loaf and this time we cut it into three equal parts, or thirds, each part is $1/3$ of the entire loaf. The three thirds equal the whole loaf, so $1/3 + 1/3 + 1/3 = 3 \times 1/3 = 1$.

If we had cut the loaf of bread so that there were two pieces but one was a third of the loaf and the other was two-thirds of the loaf, the equation would be

$$\frac{1}{3} + \frac{2}{3} = \frac{3}{3} = 1$$

If the thirds are each cut in half, we get six equal pieces, all of which add up to the original loaf. If we push two of them together we get back to a third. So $1/6 + 1/6 = 2 \times 1/6 = 1/3$. Written another way and in more detail:

$$2 \times \frac{1}{6} = \frac{2}{6} = \frac{2 \times 1}{2 \times 3} = \frac{2}{2} \times \frac{1}{3} = 1 \times \frac{1}{3} = \frac{1}{3}$$

Think of "×" as meaning "of." So half of a third being a sixth is

$$\frac{1}{2} \times \frac{1}{3} = \frac{1}{6}$$

The arithmetic of fractions, particularly addition and subtraction, is easy when we are only dealing with ones that have the same denominators such as halves (2), thirds (3), and sixths (6) as above. When the denominators are different we need to find the dreaded least common denominator (lcd).

What do we get if we push another one-sixth of a loaf of bread onto one-third of a loaf?

$$\frac{1}{6} + \frac{1}{3} = \frac{1}{6} + 2 \times \frac{1}{6} = 3 \times \frac{1}{6} = \frac{3}{6}$$

In this case, the least common denominator is simply 6 because we can always represent some number of thirds as twice that number of sixths.

What about $1/3+1/5$? We cannot easily represent thirds as fifths and so we have to subdivide each so that we have a common size. In this case, the smallest size is one-fifteenth: $1/3 = 5/15$ and $1/5 = 3/15$.

$$\frac{1}{3} + \frac{1}{5} = \frac{5}{15} + \frac{3}{15} = \frac{8}{15}$$

You might not think that fifteenths are as easy as halves, thirds, fourths, or fifths, but one fraction is as good as any other. In this example, 15 is the *least common multiple* (lcm) of 3 and 5: it is the smallest non-zero positive integer divisible by 3 and 5.

An example shows you how to compute the least common multiple of two integers. Let's work with -18 and 30. Since we care only about finding a positive integer, we can forget about the negative sign in front of 18.

Factor each number into primes: $18 = 2 \times 9 = 2 \times 3^2$ and $30 = 2 \times 3 \times 5$. We're going to gather these primes into a collection. For each prime contained in *either* factorization, put it in the collection with its exponent if it is not already there or replace what is in the collection if you find it with a larger exponent. Hence:

- The collection starts empty as { }.
- Process $18 = 2 \times 9 = 2 \times 3^2$ prime by prime.
  - 2 is not already in the collection, so insert it, yielding $\{2\}$.
  - 3 is not already in the collection with any exponent, so insert 3^2 yielding $\{2, 3^2\}$.
- Process $30 = 2 \times 3 \times 5$ prime by prime:

  - 2 is already in the collection, ignore it.
  - 3 is already in the collection with the larger exponent 2, so ignore it.
  - 5 is not in the collection, so insert it.
- The final collection is $\{2, 3^2, 5\}$.

Multiplying together all the numbers in the collection yields 90, the least common multiple. Our process ensures that each of the original numbers divides into 90 and that 90 is the smallest number for which this works.

When the numbers have no primes in common, the least common multiple is simply the absolute value of their product.

When the numerators are non-trivial (that is, not equal to 1) then we need to do some multiplication with them. As above, suppose we have found 15 to be the least common multiple of 3 and 5. Hence it is the least common denominator in $2/3 + 7/5$.

This is how we do the addition. Subtraction is similar.

$$\begin{aligned}\frac{2}{3} + \frac{7}{5} &= \frac{5}{5} \times \frac{2}{3} + \frac{3}{3} \times \frac{7}{5} \\ &= \frac{5 \times 2}{5 \times 3} + \frac{3 \times 7}{3 \times 5} \\ &= \frac{10}{15} + \frac{21}{15} \\ &= \frac{31}{15}\end{aligned}$$

To raise a rational number to a whole number exponent, raise the numerator and denominator to that exponent.

$$\left(\frac{-3}{4}\right)^5 = \frac{(-3)^5}{4^5} = \frac{-243}{1024}$$

To simplify a fraction you express it in lowest terms, which means you have factored out common primes from the numerator and denominator. To further normalize it, make it contain at most one negative sign and, if it is present, put it in the numerator.

Examples

$$\frac{1}{-2} = \frac{-1}{2} \qquad \frac{5}{5} = \frac{5^1}{5^1} = \frac{5^0}{5^0} = \frac{1}{1} = 1$$

$$\frac{2}{8} = \frac{2^1}{2^3} = \frac{2^0}{2^2} = \frac{1}{4} \qquad \frac{12}{30} = \frac{2^2 \times 3^1}{2^1 \times 3^1 \times 5^1} = \frac{2^1}{5^1} = \frac{2}{5}$$

If a prime is present in the numerator and denominator then we have that prime divided by itself, which is 1. That means we can remove it from both. This is called *cancellation*.

There is no integer strictly between 3 and 4. Given two different rational numbers, you can always find a rational number between them. Just average them: $\frac{3+4}{2} = \frac{7}{2}$.

So, while we proceed from integer to integer by adding or subtracting one, we can't slip between an integer and its successor and find another integer.

Integers are infinite in each of the positive and negative directions, and so therefore are the rational numbers, but there are an infinite number of rational numbers between any two distinct rational numbers.

Greatest common divisor

There is a more direct way of calculating the least common multiple via the greatest common divisor.

Let a and b be two non-zero integers. We can assume that they are positive. The *greatest common divisor* g is the largest positive integer such that $g|a$ and $g|b$. $g \leq a$ and $g \leq b$. We abbreviate the greatest common divisor to "gcd." One of the properties of the greatest common divisor g is that there are integers n and m so that $an + bm = g$.

If $g = 1$ then we say that a and b are coprime.

Given this,

$$\operatorname{lcm}(a, b) = \frac{a\, b}{\gcd(a, b)}.$$

If either a or b is negative, use its absolute value.

To calculate $\gcd(a, b)$ we use quotients and remainders in *Euclid's algorithm*. By the properties of division for positive integers a and b with $a \geq b$ there exist non-negative integers q and r such that

$$a = bq + r$$

with $0 \leq r < b$. q is called the *quotient* upon dividing a by b and r is the *remainder*. Because $r < b$, q is as large as it can be. If $r = 0$ then $b|a$ and $\gcd(a, b) = b$.

So let's suppose that n divides a and b. Then n divides $a - bq = r$. In particular, for $n = \gcd(a, b)$ then

$$\gcd(a, b) = \gcd(b, r).$$

We have replaced the calculation of the gcd of a and b with the calculation of the gcd of b and r, a smaller pair of numbers. We can keep repeating this process, getting smaller and smaller pairs. Since $r \geq 0$, we eventually stop.

Once we get an r that is 0, we go back and grab the previous remainder. That is the gcd.

Question 3.4.1

Compute $\gcd(15295, 38019)$. Factor the answer if you can.

Euclid originally used subtractions but with modern computers we can efficiently compute the quotients and remainders.

3.4.2 Getting formal again

Let's rewind now and look at the rational numbers and their operations in the ways we did with the natural numbers, whole numbers, and integers.

Just as we introduced $-w$ for a whole number w to be the unique integer value such that $w + -w = 0$, we set $1/w = \frac{1}{w}$ to be the unique value such that $1/w \times w = 1$ for non-zero w. $1/w$ is called the *inverse* of w. Technically, $1/w$ is the unique multiplicative inverse of w but that's quite a mouthful. If I don't state that w is non-zero, assume it is.

The rational numbers extend the integers with a multiplicative inverse and similarly extended rules for addition, subtraction, multiplication, and division.

For any two integers a and non-zero b we define $a/b = \frac{a}{b}$ to be $a \times \frac{1}{b}$. $\frac{b}{b} = 1$.

Two expressions of the form $\frac{c \times a}{c \times b}$ and $\frac{a}{b}$ for non-zero c and b refer to the very same rational number.

If a and b have no prime factors in common and b is positive, we say the rational number is shown in lowest terms. Simplifying a rational number expression means to write it in its lowest terms. We can now write out the rules for arithmetic.

Equality

Two expressions $\frac{a}{b}$ and $\frac{c}{d}$ with non-zero b and d represent the same rational number if $a \times d = c \times b$.

Addition

Process: Cross-multiply the numerators and denominators and add the results to get the new numerator, multiply the denominators to get the new denominator, simplify.

$$\frac{a}{b} + \frac{c}{d} = \frac{a \times d + c \times b}{b \times d}$$

for non-zero b and d. Alternatively, convert each fraction to have the same least common denominator, add the numerators, and simplify.

Subtraction

Process: Cross-multiply the numerators and denominators and subtract the results to get the new numerator, multiply the denominators to yield the new denominator, simplify.

$$\frac{a}{b} - \frac{c}{d} = \frac{a \times d - c \times b}{b \times d}$$

for non-zero b and d. Alternatively, convert each fraction to have the same least common denominator, subtract the numerators, and simplify.

Negation

Negating a value is the same as multiplying it by -1. Negative signs "cancel" across the numerator and denominator.

$$-\frac{a}{b} = \frac{-a}{b} = \frac{-1 \times a}{b} = \frac{a}{-1 \times b} = \frac{a}{-b}$$

for non-zero b.

Multiplication

Process: Multiply the numerators to yield the new numerator, multiply the denominators to yield the new denominator, simplify.

$$\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d}$$

for non-zero b and d.

Inversion

The inverse of a non-zero rational number is the rational number formed by swapping the numerator and denominator.

$$\frac{1}{\left(\frac{a}{b}\right)} = \left(\frac{a}{b}\right)^{-1} = \frac{b}{a}$$

for non-zero a and b. Raising a rational number to the -1 power means to compute its inverse.

Division

Divide two rational numbers by multiplying the first by the inverse of the second.

$$\frac{\left(\frac{a}{b}\right)}{\left(\frac{c}{d}\right)} = \frac{a}{b} \times \frac{1}{\left(\frac{c}{d}\right)} = \frac{a}{b} \times \frac{d}{c} = \frac{a \times d}{b \times c}$$

for non-zero b, c, and d.

Exponentiation

Similar to other numbers, raising a rational number to the 0th power yields 1. Raising it to a negative integer power means to swap the numerator and denominator and raise each to the absolute value of the exponent.

$$\left(\frac{a}{b}\right)^n = \begin{cases} \dfrac{a^n}{b^n} & \text{for integer } n > 0 \\ 1 & \text{for } n = 0 \\ \dfrac{b^{-n}}{a^{-n}} & \text{for integer } n < 0 \end{cases}$$

for non-zero b or if n is a negative integer, non-zero a.

The rational numbers $\mathbb{Q}$ is the infinite ordered extension of $\mathbb{Z}$ formed by appending the multiplicative inverses $\frac{1}{n}$ of all non-zero integers n, and then further extending by defining values $\frac{n}{m} = n \times \frac{1}{m}$ for integers n and non-zero m.

The rational numbers have a unique identity element $\mathbf{1}$ such that for any non-zero r there exists a unique multiplicative inverse $\frac{1}{r}$ such that $r \times \frac{1}{r} = \mathbf{1}$. $\mathbb{Q}$ is closed under the commutative addition and multiplication operations, subtraction, and under division by non-zero values.

The rational numbers seem to finally solve most of our problems about doing arithmetic and getting a valid answer. However, even though $\sqrt{4}$ and $\sqrt{\frac{1}{25}}$ are both rational numbers, neither $\sqrt{2}$ nor $\sqrt{\frac{1}{5}}$ is.

If $\sqrt{2}$ were rational, there would exist positive integers m and n such that $m/n = \sqrt{2}$, meaning that $\dfrac{m^2}{n^2} = 2$.

We can assume m and n have no factors in common. This is key!

Let's show this is not possible. Every even integer is of the form $2k$ for some other integer k. Similarly, all odd integers are of the form $2k + 1$. Therefore the square of a integer looks like $4k^2$, which is even, and the square of an odd integer looks like $4k^2 + 2k + 1$, which is odd.

This also shows that if an even integer is a square then it is the square of an even integer. If an odd integer is a square then it is the square of an odd integer.

If $\frac{m^2}{n^2} = 2$ then $m^2 = 2n^2$ and so m^2 and therefore m are even integers. So there is some integer j such that $m = 2j$ and $m^2 = 4j^2$.

We then have

$$m^2 = 4j^2 = 2n^2$$

or

$$2j^2 = n^2.$$

As before, this shows that n^2 and n are *even*. So both m and n are even and they share the factor 2.

But we assumed m and n have no factors in common, a contradiction! Thus there exist no such m and n and $\sqrt{2}$ *is not a rational number*.

Question 3.4.2

By similar methods, show $\sqrt{3}$ is not rational.

3.5 Real numbers

When we're were done looking at the real numbers we'll have finished analyzing the typical numbers most people encounter. Let's begin with decimals.

3.5.1 Decimals

A *decimal* expression for a real number looks like

- an optional minus sign,
- followed a finite number of digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9,
- followed by a period, also called the *decimal point*,
- followed by a finite or infinite number of digits.

In many parts of the world, the decimal point is a comma instead of a period, but I use the United States and UK convention here.

If there are no digits after the decimal point then the decimal point may be omitted.

Any trailing 0s on the right are usually omitted when you are using the number in a general mathematical context. They may be kept in situations where they indicate the precision of a measurement or a numeric representation in computer code.

Any leading 0s on the left are usually omitted. We have

$$\begin{aligned} 0 &= 0. = .0 = 000.00 \\ 1 &= 1. = 1.0 = 000001 \\ -3.27 &= -03.27 = -3.27000000000 \end{aligned}$$

By convention it is common to have at least a single 0 before the decimal point and a single 0 after: 0.0 and -4.0, for example.

The integer 1327 is a shorthand way of writing

$$1 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 7 \times 10^0.$$

Similarly, the integer -340 is

$$(-1)\left(3 \times 10^2 + 4 \times 10^1 + 0 \times 10^0\right).$$

We extend to the right of the decimal point by using negative powers of 10:

$$\begin{aligned} 13.27 &= 1 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 7 \times 10^{-2} \\ -0.340 &= (-1)\left(0 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 0 \times 10^{-3}\right) \end{aligned}$$

The decimal point is at the place where we move from 10^0 to 10^{-1}.

Since 10^{-1} is $\frac{1}{10}$ = one tenth, the digit immediately after the decimal point is said to be in the "tenths position." The one after that is the "hundredths position" because it corresponds to $10^{-2} = \frac{1}{100}$ = one hundreth. We continue this way to the thousandth, ten-thousanth, hundred-thousandth, millionth positions, and so on.

To convert a fraction like $1/2$ to decimal, we try to re-express it with denominator equal to some power of ten. In this case it is easy because $1/2 = 5/10$. So five-tenths is 0.5.

For $3/8$ we need to go all the way up to $375/1000$.

$$\begin{aligned}
\frac{3}{8} &= \frac{3}{2^3} \times \frac{5^3}{5^3} \\
&= \frac{3}{2^3} \times \frac{125}{125} \\
&= \frac{375}{1000} \\
&= \frac{300}{1000} + \frac{70}{1000} + \frac{5}{1000} \\
&= \frac{3}{10} + \frac{7}{100} + \frac{5}{1000} \\
&= 3 \times 10^{-1} + 7 \times 10^{-}2 + 5 \times 10^{-3} \\
&= .375
\end{aligned}$$

Since $10 = 2 \times 5$, each power of 10 is the product of the same power of 2 times the same power of 5. That's why we chose 5^3 in the above example: $10^3 = 2^3 5^3$.

This method doesn't always work to convert a fraction to a decimal. The decimal expression of $1/7$ is

$$0.142857142857142857142857142857142857142857\ldots$$

Notice the section "142857" that repeats over and over.

$$0.\boxed{142857}\,142857\ 142857\ 142857\ 142857\ 142857\ 142857\ldots$$

It goes on repeating forever, block after adjacent block. This is an infinite decimal expansion. We write the repeating block with a line over it:

$$\frac{1}{7} = 0.\overline{142857}$$

Any rational number has a finite decimal expression or an infinite one with a repeating block.

We showed above how to go from a finite decimal expansion to a fraction: express the decimal as a sum of powers of 10 and then do the rational number arithmetic.

$$\begin{aligned} 2.13 &= 2 \times 10^0 + 1 \times 10^{-1} + 3 \times 10^{-2} \\ &= 2 + \frac{1}{10} + \frac{3}{100} \\ &= \frac{213}{100} \end{aligned}$$

This is already simplified but in general we need to do that at the end.

The general process is slightly more complicated. Let $r = 0.\overline{153846}$. The repeating block has **6** digits and it is immediately after the decimal point. Multiply both sides by $10^{\mathbf{6}} = 1000000$:

$$1000000r = 153846.\overline{153846}$$

and so

$$1000000r - r = 153846.\overline{153846} - 0.\overline{153846}$$

which gives

$$\begin{aligned} 999999r &= 153846 \\ r &= \frac{153846}{999999} \\ r &= \frac{2}{13} \end{aligned}$$

Question 3.5.1

How would you adjust this if the repeating block begins more to the right of the decimal point?

If the entire decimal expression does not repeat, you can separate it into a finite expansion plus the repeating one divided by the appropriate power of 10.

$$3.2\overline{153846} = 3.2 + 0.0\overline{153846}$$

$$
\begin{aligned}
&= \frac{32}{10} + \frac{2}{13} \times 10^{-1} \\
&= \frac{32}{10} + \frac{2}{130} \\
&= \frac{32}{10} \times \frac{13}{13} + \frac{2}{130} \\
&= \frac{416}{130} + \frac{2}{130} \\
&= \frac{418}{130} = \frac{209}{65}
\end{aligned}
$$

These calculations show the relationships between rational numbers and their decimal expansions.

Question 3.5.2

What is the rational number corresponding to $0.\overline{9}$?

If r is a real number then $\lfloor r \rfloor$, the *floor* of r, is the largest integer $\leq r$. Similarly, $\lceil r \rceil$, the *ceiling* of r, is the smallest integer $\geq r$.

3.5.2 Irrationals and limits

The case we have not considered is an infinite decimal expansion that does not have an infinitely repeating block. Not being rational, it is called an *irrational number*. The real numbers are the rational numbers in addition to all the irrational numbers. Since $\sqrt{2}$ is not rational, it must be irrational.

Let's consider approximation of a real number by a decimal.

$$\pi = 3.14159265358979323846264338327950\ldots$$

is an irrational number and so has no infinitely repeating blocks. It is not $22/7$ and it is not 3.14. Those are rational and decimal approximations to π, and not even very good ones.

π exists as a number even though you cannot write it down as a fraction or write out the infinite number of digits that express it. π is in $\mathbb{R}$ but it is not in $\mathbb{Q}$.

Consider

$$3.1 \rightarrow 3.14 \rightarrow 3.141 \rightarrow 3.1415 \rightarrow 3.14159 \rightarrow 3.141592 \rightarrow 3.1415926 \rightarrow \cdots$$

This is a sequence of rational numbers (expressed as decimals) that get closer and closer to the actual value of π.

Want to be within one-millionth of the actual value? $\pi - 3.1415926 < 0.000001$. Within one-hundred-millionth? $\pi - 3.141592653 < 0.00000001$. We could keep going.

We have a sequence of rational numbers such that if we set a closeness threshold like one-millionth then one member of the sequence and all that follow it are at least that close to π. We say the irrational number π is the *limit* of the given sequence of rational numbers.

If you make the threshold smaller, we can find a possibly later sequence member so we will be at least that close from then on. We say the above sequence *converges* to π.

Think about this. All the members of the sequence are rational numbers but the limit is not. Informally, if we take the rational numbers and throw in all the limits of convergent sequences of them, we get the real numbers.

Of course, there are sequences of rational numbers that converge to rational numbers.

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{n}, \ldots$$

converges to the limit 0. Here we let n get larger and larger and we write

$$\lim_{n \to \infty} \frac{1}{n} = 0$$

Similarly,

$$-\frac{2}{1}, -\frac{3}{2}, -\frac{4}{3}, \ldots, -\frac{n+1}{n}, \ldots$$

converges to -1. For a non-obvious example, consider

$$\lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

Even though n is getting bigger, the expression inside the parentheses is getting closer to 1. As n gets bigger the computed values appear to converge.

$$\begin{aligned} n = 1 &\to 1.5 \\ 10 &\to 2.5937424601\ldots \\ 10000 &\to 2.71814592682\ldots \\ 100000 &\to 2.71826823719\ldots \end{aligned}$$

This sequence converges to $e = 2.718281828459045235360\ldots$, the base of the natural logarithms and an irrational number. Like π, e is a special value in mathematics and shows up "naturally" in many contexts.

The sequence

$$1, 2, 3, 4, 5, \ldots, n, \ldots$$

does not converge to any finite rational number. We call it a *divergent sequence*.

We define the real numbers to be the extension of $\mathbb{Q}$ that contains the limits of all convergent sequences of rational numbers. Furthermore, the real numbers are closed under taking the limits of convergent sequences of real numbers.

Closure, closure, closure. The concept really is fundamental to the kinds of numbers we use every day.

Limits are essential in calculus. The idea that we can have an infinite sequence of numbers that converges to a fixed and unique value is unlike anything most people have seen earlier in their mathematical studies. It can be a scary and daunting concept at first.

Remember above when I asked you to compute $0.\overline{9}$? This is the limit of the sequence

$$\begin{array}{l} 0.9 \\ 0.99 \\ 0.999 \\ 0.9999 \\ 0.99999 \\ \quad\ldots \end{array}$$

Every time we move to the next member of the sequence we add another 9 on the far right. It appears the limit is 1. If you want to be within one quadrillionth of $1 = 10^{-15}$, then go out to 0.9999999999999999. However close you want to be to 1, I can add enough 9s to the end so I am at least that close and every member of the sequence after it is too. The sequence converges to 1, its limit.

We can also consider sequences where each member is a sum that builds on the previous member. This sequence

$$\begin{array}{l} 1 \\ 1 - \frac{1}{3} \\ 1 - \frac{1}{3} + \frac{1}{5} \\ 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \\ 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \\ \quad \ldots \end{array}$$

converges to $\frac{\pi}{4}$ but it does so excruciatingly slowly. For use with computer calculations, it's critical to find sequences that converge quickly.

3.5.3 Binary forms

Just as we can write a whole number in base 10 form using the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, we can also use only the bits 0 and 1 to represent it in binary form. We saw examples of this in section 2.2.

This algorithm converts from decimal to binary for w in $\mathbb{W}$:

1. If $w = 0$ then the result is also 0, and we are done.
2. Otherwise, let b be an initially empty placeholder where we put the bits.
3. If w is odd, put a 1 to the left of anything in b. Set w to $w - 1$. Otherwise, put a 0 to the left of anything in b. In both cases, now set w to $w/2$.
4. If $w = 0$, we are done and b is the answer. Otherwise, go back to step 3.

For example, let $w = 13$. Initially b is empty.

- w is odd, so b is now 1 and we set $w = (w - 1)/2 = 6$.
- w is even, so b is now 01 and we set $w = w/2 = 3$.
- w is odd, so b is now 101 and we set $w = (w - 1)/2 = 1$.
- w is odd, so b is now 1101 and we set $w = (w - 1)/2 = 0$.
- $w = 0$ and we are done. The representation of w in binary is $b = 1101_2$.

We put the subscript 2 at the end of the number to remind ourselves that we are in base 2.

Now suppose we start with a decimal r with $0 \leq r < 1$. We want to come up with a base 2 representation using just 0s and 1s to the right of the "binary point." We expand as we usually do but instead of using negative powers of 10, we use negative powers of 2.

$$.011_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

In base 10, this is $\frac{1}{4} + \frac{1}{8} = \frac{3}{8} = .375$.

Our algorithm for converting the fractional part of a real number to binary is reminiscent of the one above.

1. If $r = 0$ then the result is also 0, and we are done.
2. Otherwise, let b be a placeholder containing only "." where we put the bits.
3. Multiply r by 2 to get s. Since $0 \leq r < 1$, $0 \leq s < 2$. If $s \geq 1$, put a 1 to the right of anything in b and set $r = s - 1$. Otherwise, put a 0 to the right of anything in b and set $r = s$.
4. If $r = 0$, we are done and b is the answer. Otherwise, go back to step 3.

This sounds reasonable. Let's try it out with $r = .375_{10}$ to confirm our example above. r is not 0 and b starts with the binary point ".".

- Set $s = 2r = .75$, which is less than 1. We append a 0 to the right in b and set $r = s = .75$. b is now .0.
- Set $s = 2r = 1.5$, which is greater than or equal to 1. Append a 1 to the right in b and set $r = s - 1 = .5$. b is now .01.
- Set $s = 2r = 1$, which is greater than or equal to 1. Append a 1 to the right in b and set $r = s - 1 = 0$. b is now .011.
- Since $r = 0$, we are done and the answer is $.011_2$.

In a less verbose table form, this looks like

s	b	r
	.	.375
.75	.0	.75
1.5	.01	.5
1	.011	0

The first line holds the initial settings. The answer is b on the last line where $r = 0$.

The answer agrees with the previous example. Let's do another with $r = .2_{10}$.

s	b	r
	.	.2
.4	.0	.4
.8	.00	.8
1.6	.0001	.6
1.2	.00011	.2
.4	.000110	.4
$\vdots$	$\vdots$	$\vdots$

I stopped because the process has started to repeat itself. Put another way, and using the notation we employed for repeating decimals,

$$.2_{10} = .\overline{0011}_2$$

There is no exact finite binary expansion for the decimal 0.2 but it repeats in a block.

The following hold in parallel to the decimal case:

- A base 10 rational number has either a finite binary expansion or it repeats in blocks.
- A repeating block binary expansion is a base 10 rational number.
- An irrational real number has an infinite, non-repeating-block binary expansion.
- A binary expansion that has no repeating blocks is irrational.

Given a real value with whole number part w and decimal part $r < 1$, you create the full binary form by concatenating the binary forms of each. The full binary expansion of the decimal 5.125 is 110.001_2, for example.

Question 3.5.3

What is the binary expansion of 17.015625_{10}? Of $\frac{4}{3}$?

3.5.4 Continued fractions

There is yet another expansion for real numbers that is usually not taught in high school algebra classes. This is the *continued fraction* and two examples are on the right-hand sides in each of the following:

$$\tfrac{15}{11} = 1\tfrac{4}{11} = 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{3}}} \qquad \tfrac{11}{15} = 0 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{3}}}}$$

We write the integer portion out front and then construct a recurring sequence of fractions with 1 in the numerators.

Working through the first example shows you the algorithm. Begin with writing the integer portion out front.

First approximation: 1

What's left is $\frac{4}{11}$. Invert this to get $\frac{11}{4} = 2\frac{3}{4}$. Take the whole number part and use this as the second part of the expansion.

Second approximation: $1 + \dfrac{1}{2}$

Invert the remaining fractional part to get $\frac{4}{3} = 1\frac{1}{3}$. The whole number part goes into the expansion.

Third approximation: $1 + \cfrac{1}{2 + \cfrac{1}{1}}$

Invert $\frac{1}{3}$ to get 3 for the expansion. There is no non-zero fractional part and we are done.

Final expansion: $1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{3}}}$

This is a finite continued fraction expansion since there are only a finite number of terms. Working through the fraction arithmetic, if you start with a finite continued fraction you end up with a rational number. What we are doing here is just a variation on Euclid's algorithm from subsection 3.4.1.

Even more interesting, every rational number terminates in this way when you do the expansion. We don't have to worry about repeating blocks for rational numbers converted to continued fractions as we do with decimal and binary expansions.

Question 3.5.4

What is the continued fraction expansion of $-\frac{97}{13}$? Of 0.375?

Using variable names, we can write a finite continued fraction as

$$b_0 + \cfrac{1}{b_1 + \cfrac{1}{b_2 + \cfrac{1}{\ddots + \cfrac{1}{b_n}}}}$$

Here all the b_j are in $\mathbb{Z}$ and $b_j > 0$ for $j > 0$. That is, b_0 can be negative but the rest must be positive integers. An alternative and much shorter notation for the above is

$$[b_0; b_1, b_2, \ldots, b_n].$$

It is also possible to represent a rational number using the form with one more term

$$[b_0; b_1, b_2, \ldots, b_{n-1}, b_n - 1, 1]$$

but I prefer the shorter version. If we decide that the last term cannot be 1, there is a unique representation for a rational number.

Question 3.5.5

Let $r > 0$ be in $\mathbb{R}$. Is

$$r[b_0; b_1, b_2, \ldots, b_n] = [rb_0; rb_1, rb_2, \ldots, rb_n]?$$

What if we are given an expansion that is infinite? It can't be a rational number and it does, in fact, converge to an irrational real number.

Every irrational real number has a unique infinite continued fraction expansion $f = [b_0; b_1, b_2, b_3, \ldots]$.

Infinite continued fractions can have repeating blocks or blocks that repeat via a formula. A line over a block means that it repeats, as usual.

The first expansion in the following table is the “golden ratio” while the last is for e, the base of the natural logarithms. In the expansion for e, note how the integer between the two 1s is incremented by 2 in every block.

Value	**Expansion**
$\frac{1+\sqrt{5}}{2}$	$[1; \overline{1}]$
$1 + \sqrt{2}$	$[2; \overline{2}]$
$\frac{3+\sqrt{13}}{2}$	$[3; \overline{3}]$
$\sqrt{3}$	$[1; \overline{1, 2}]$
$\sqrt{7}$	$[2; \overline{1, 1, 1, 4}]$
$\tan(1)$	$[1; 1, 1, 3, 1, 5, 1, 7, 1, 9, 1, 11, \ldots]$
e	$[2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, \ldots]$

Question 3.5.6

Calculate the first 6 digits of the golden ratio from its continued fraction.

Let’s revisit the first two examples, which I now write in short form:

$$\tfrac{15}{11} = [1; 2, 1, 3] \qquad\qquad \tfrac{11}{15} = [0; 1, 2, 1, 3].$$

What do you notice about these? First, the numbers are reciprocals and, second, they have the same expansion except the latter one has a 0 at the beginning. This is true in general.

Let r be in $\mathbb{Q}$ and positive. Suppose $r < 1$ and its continued fraction expansion is

$$[0; b_1, b_2, \ldots, b_n].$$

The expansion for $\frac{1}{r}$ is

$$[b_1; b_2, \ldots, b_n].$$

On the other hand, if $r \geq 1$ with expansion

$$[b_0; b_1, b_2, \ldots, b_n]$$

then the expansion for $\frac{1}{r}$ is

$$[0; b_0, b_1, b_2, \ldots, b_n]$$

Given an infinite continued fraction $f = [b_0; b_1, b_2, b_3, \ldots]$, it's natural to look at the sequence of finite fractions, the *convergents* of f,

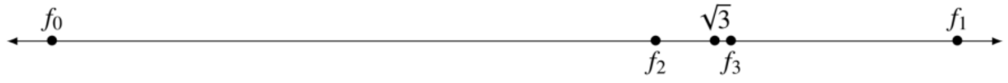
$$\begin{aligned} f_0 &= [b_0;] = \frac{x_0}{y_0} \\ f_1 &= [b_0; b_1] = \frac{x_1}{y_1} \\ f_2 &= [b_0; b_1, b_2] = \frac{x_2}{y_2} \\ &\vdots \\ f_n &= [b_0; b_1, b_2, b_3, \ldots, b_n] = \frac{x_n}{y_n} \end{aligned}$$

and ask about the relationship between f and the f_j. Each x_j is an integer and each y_j is a positive integer. The f_j are expressed in reduced form. (That is, $\frac{1}{2}$ and not $\frac{3}{6}$.)

The convergents f_j have the following properties with respect to a specific convergent f_n:

- $f_1 > f_3 > f_5 > \cdots > f_n$ for all f_j with odd $j < n$.
- $f_2 < f_4 < f_6 < \cdots < f_n$ for all f_j with even $j < n$.
- If $j < k < n$ then $|f_n - f_k| < \left|f_n - f_j\right|$.

This means the convergents oscillate above and below f_n, always getting closer.



This example shows the rapid convergence to $\sqrt{3}$.

With f as above,

$$f_2 = [b_0; b_1, b_2] = b_0 + \cfrac{1}{b_1 + \cfrac{1}{b_2}} = \frac{b_0 b_1 b_2 + b_2 + b_0}{b_1 b_2 + 1} = \frac{b_2(b_0 b_1 + 1) + b_0}{b_1 b_2 + 1}.$$

Question 3.5.7

Compute f_1, x_1, and y_1, and f_3, x_3, and y_3. Establish a guess about how to compute the x_n and y_n given the values for $n-1$ and $n-2$. Confirm if this works for f_4, x_4, and y_4.

Convergence Properties of Continued Fractions [3]

Let r in $\mathbb{R}$ be the value of the infinite continued fraction $f = [b_0; b_1, b_2, b_3, \ldots]$. Let $f_j = \frac{x_j}{y_j}$ be the convergents.

- Each convergent is a reduced fraction. That is, $\gcd(x_j, y_j) = 1$.
- If $k > j$ then $y_k > y_j$.
- The denominators y_j are increasing exponentially:

$$y_j \geq 2^{\frac{j-1}{2}}$$

- We can approximate r as closely as we wish by computing a divergent with large enough j.

$$\left|r - f_j\right| = \left|r - \frac{x_j}{y_j}\right| < \frac{1}{y_j y_{j+1}}$$

Question 3.5.8

What do you need to modify in the preceding statements so they hold for a finite continued fraction?

To learn more

Continued fractions are a fascinating but somewhat specialized area of mathematics. They're not difficult but they are not used in every field. The topic is often covered briefly in algebra and number theory texts but there are only a few dedicated books about them. [7, Chapter 10] [3] [8]

3.6 Structure

I took some time to show the operations and the properties of the real numbers and its subsets like the integers and rational numbers because these are very common in other parts of mathematics when properly abstracted. This structure allows us to learn and prove things and then apply them to new mathematical collections as we encounter them. We start with three: *groups*, *rings*, and *fields*.

3.6.1 Groups

Consider a collection of objects which we call $\mathbf{G}$. For example, $\mathbf{G}$ might be $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$ as above. We also have some pairwise operation between elements of $\mathbf{G}$ we denote by "$\circ$". It's a placeholder for an action that operates on two objects.

This "$\circ$" operation could be addition "+" or multiplication "$\times$" for numbers, but might be something entirely different. Use your intuition with numbers, but understand that the general case is, well, more general. We call the collection together with its operation $(\mathbf{G}, \circ)$.

We write the use of "$\circ$" the same as we would normally with addition or multiplication, between the elements. We write $a \circ b$ for a and b in $\mathbf{G}$.

This is called *infix* notation. Negation like -7 uses *prefix* notation. The factorial operation $n! = 1 \times 2 \times \cdots \times (n-1) \times n$ uses *postfix* notation.

We say $(\mathbf{G}, \circ)$ is a group if the following conditions are met

- If a and b are in $\mathbf{G}$ then $a \circ b$ is in $\mathbf{G}$. This is closure.
- If a, b, and c are in $\mathbf{G}$ then $(a \circ b) \circ c = a \circ (b \circ c)$ is in $\mathbf{G}$. This is associativity.
- There exists a unique element id in $\mathbf{G}$ such that $a \circ id = id \circ a = a$ for every a in $\mathbf{G}$. This is the existence of a unique identity element.
- For every a in $\mathbf{G}$, there is an element denoted a^{-1} such that $a^{-1} \circ a = a \circ a^{-1} = id$. This is the existence of a inverse.

The inverse is unique. Suppose there are two elements b and c such $b \circ a = a \circ b = id$ and $c \circ a = a \circ c = id$. Then $b \circ a \circ c = id \circ c$ by applying "$\circ c$" on the right. So $b \circ (a \circ c) = c$. Since c is an inverse of a, $b = c$.

We do not require the "$\circ$" operation to be commutative: $a \circ b$ need not equal $b \circ a$. When this is true for all a and b, we call $\mathbf{G}$ a commutative group.

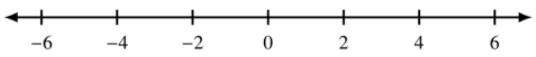
In the mathematical literature, commutative groups are called *abelian groups* in honor of the early nineteenth century mathematician Niels Henrik Abel, but we stick to the descriptive name.

While you are likely aware of the quadratic formula for finding the roots of a polynomial like $x^2 + x - 6$, you may not know that there are also (very messy) formulas for third and fourth degree polynomials. Despite others working on this for hundreds of years, Abel finally proved that there is no corresponding formula for polynomials of degree 5.

A subset of $\mathbf{G}$ that uses the same operation "$\circ$" and is closed under it, contains id, and is closed under inversion is called a *subgroup* of $\mathbf{G}$.

Examples

- The natural numbers are not a group under addition because of the lack of 0 and negative numbers.
- The whole numbers are not a group under addition because all positive numbers lack their negative counterparts.
- The integers $\mathbb{Z}$, rational numbers $\mathbb{Q}$, and real numbers $\mathbb{R}$ are each a group under addition with identity element 0. $\mathbb{Z}$ is a subgroup of $\mathbb{Q}$, which is a subgroup of $\mathbb{R}$.
- The even integers are a group under addition with identity element 0. They are a subgroup of $\mathbb{Z}$.



- The odd integers are not a group under addition.
- The integers $\mathbb{Z}$ are not a group under multiplication because most integers lack multiplicative inverses.
- The rational numbers $\mathbb{Q}$ are not a group under multiplication because there is no multiplicative inverse for 0.
- The rational numbers without 0 are a group under multiplication but not under addition.
- Similarly, the non-zero real numbers are a group under multiplication but not under addition.

Much of the fuss we made when we proceeded from $\mathbb{W}$ to $\mathbb{N}$ to $\mathbb{Z}$ was to systematically examine their properties to ultimately show $\mathbb{Z}$ is a group under "+".

In all the examples above where we have groups, they are commutative groups. The study of groups, finite and infinite, commutative and non-commutative, is a fascinating and essential topic at the core of much of mathematics and physics.

For another example, imagine you live on a world that is one long straight infinite street.

For our group, take movements of the form "walk 12 meters to the right" and "walk 4 meters to the left." The group operation is composition "∘" thought of as "and then." We can write

$$\begin{aligned}
a &= \text{“walk 12 meters to the left”}\\
b &= \text{“walk 4 meters to the right”}\\
a \circ b &= \text{“walk 12 meters to the left” } \textit{and then} \text{ “walk 4 meters to the right”}\\
&= \text{“walk 8 meters to the left”}
\end{aligned}$$

Note I didn't specify where on the street to start: all movements are relative. The inverse of a "walk to the right" element is the corresponding "walk to the left" element. The identity element *id* is "walk to the right 0 meters" which we take to be the same as "walk to the left 0 meters."

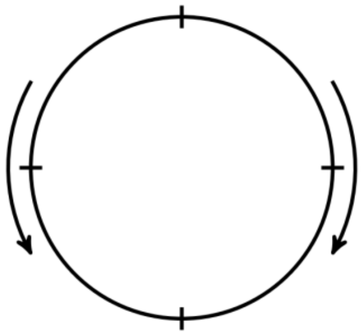
Verify for yourself that "∘" is associative and commutative.

You can extend this group in two dimensions by adding similar "walk forward" and "walk backward" elements. For three dimensions you would indicate up and down movements. Think about associativity and commutativity in each case.

Question 3.6.1

Is this a finite or infinite group? What are some subgroups?

Rather than being on a straight line, consider being on a circle of diameter 4 meters. You are only able to move in natural number meter increments clockwise or counterclockwise.



We can equate moving 4 meters in either direction with moving 0 meters since we move all the way around the circle. Moving 5 meters counterclockwise is the same element as moving 1 meter in that direction or 3 meters in the other.

Question 3.6.2

Is this a finite or infinite group? If we allow moving clockwise or counterclockwise in any non-negative real number increments, is it a finite or infinite group?

3.6.2 Rings

When we have more than one operation, we get a more sophisticated structure called a *ring* if certain requirements are met. For convenience we call these two operations "+" and "×" but remember they could behave in very different ways from the addition and multiplication we have for numbers.

We say (**R**, +, ×) is a *ring* if the following conditions are met

- **R** is a commutative group under "+" with identity element 0.
- If a, b, and c are in **R** then $(a \times b) \times c = a \times (b \times c)$ is in **R**. This is associativity for "×".
- There exists an element 1 in **R** such that $a \times 1 = 1 \times a = a$ for every a in **R**. This is the existence of a multiplicative identity element.
- Multiplication "×" distributes over addition "+". If a, b, and c are in **R** then $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a)$.

Note that $0 \neq 1$.

While addition is commutative in a ring, multiplication need not be. As you might guess, a *commutative ring* is one with commutative multiplication. A *non-commutative ring* has the obvious definition.

A subgroup of **R** under "+" is a *subring* if it also shares the same "×", contains 1, and is closed under "×".

Examples

- The integers $\mathbb{Z}$, rational numbers $\mathbb{Q}$, and real numbers $\mathbb{R}$ are each a ring under addition and multiplication with identity elements 0 and 1, respectively. $\mathbb{Z}$ is a subring of $\mathbb{Q}$ which is a subring of $\mathbb{R}$. When we want to consider $\mathbb{Z}$ as an additive group, we write $\mathbb{Z}^+$.
- The even integers are not a subring of $\mathbb{Z}$ because they do not contain 1.
- Consider all elements of $\mathbb{R}$ of the form $a + b\sqrt{2}$ for a and b integers. We have

$$\begin{aligned}
\mathbf{0} &= 0 + 0\sqrt{2} \\
\mathbf{1} &= 1 + 0\sqrt{2} \\
-\left(a + b\sqrt{2}\right) &= -a - b\sqrt{2} \\
\left(a + b\sqrt{2}\right) + \left(c + d\sqrt{2}\right) &= (a + c) + (b + d)\sqrt{2} \\
\left(a + b\sqrt{2}\right) \times \left(c + d\sqrt{2}\right) &= (ac + 2bd) + (ad + bc)\sqrt{2}
\end{aligned}$$

  We call this $\mathbb{Z}\left[\sqrt{2}\right]$ and it is a commutative ring that extends $\mathbb{Z}$. Other than $\mathbb{Q}$, this is the first ring we have seen that is larger than $\mathbb{Z}$ but smaller than $\mathbb{R}$ itself.

In some commutative rings it is possible for $a \times b = 0$ with neither a nor b being 0. $\mathbf{R}$ is called an *integral domain* if this is not possible. Said otherwise, we must have a or b being 0 for the product to be 0. All the rings we have seen so far are integral domains.

3.6.3 Fields

A *field* $\mathbf{F}$ is a commutative ring where every non-zero element has a multiplicative inverse. A field is closed under division by non-zero elements.

$\mathbb{Q}$ and $\mathbb{R}$ are fields but $\mathbb{Z}$ is not. $\mathbb{Q}$ is a *subfield* of $\mathbb{R}$. Viewed from the opposite direction, $\mathbb{R}$ is an *extension* of $\mathbb{Q}$. For example, if we look at all numbers of the form $r + \sqrt{2}s$ with r and s in $\mathbb{Q}$, and perform the arithmetic operations in the usual way, we have an extension field of $\mathbb{Q}$ that is a subfield of $\mathbb{R}$. We denote this field by $\mathbb{Q}[\sqrt{2}]$.

All fields are integral domains. Suppose otherwise and $a \times b = 0$ with neither a nor b being 0. Then there exists a^{-1} such that $a^{-1} \times a = 1$. Hence $a^{-1} \times a \times b = a^{-1} \times 0$ means $1 \times b = 0$. But we said b is not 0! We have a contradiction and there can be no such b. So the field is an integral domain.

3.6.4 Even greater abstraction

Though we do not need them in the rest of this book, I want to name two additional algebraic structures since we have already seen examples of them. If $\mathbb{Z}$ is a group under addition, what is $\mathbb{W}$? It doesn't contain the additive inverses like -2 and so it can't be a group.

Even worse, what is $\mathbb{N}$? Here we don't even have 0, the additive identity element.

We say $(\mathbf{G}, \circ)$ is a *semigroup* if the following conditions are met:

- If a and b are in $\mathbf{G}$ then $a \circ b$ is in $\mathbf{G}$. This is closure.
- If a, b, and c are in $\mathbf{G}$ then $(a \circ b) \circ c = a \circ (b \circ c)$ is in $\mathbf{G}$. This is associativity.

$(\mathbf{G}, \circ)$ is a *monoid* if we also have:

- There exists a unique element id in $\mathbf{G}$ such that $a \circ id = id \circ a = a$ for every a in $\mathbf{G}$. This is the existence of a unique identity element.

With this, $\mathbb{N}$ is a semigroup and $\mathbb{W}$ is a monoid. All groups are monoids and all monoids are semigroups.

In the summary in section 3.10, I've provided a table and chart showing how these algebraic structures are related to each other and the collections of numbers we are working through in this chapter.

To learn more

Group theory is ubiquitous across many areas of mathematics and physics. [10] Rings, fields, and other structures are fundamental to areas of mathematics like algebra, algebraic number theory, commutative algebra, and algebraic geometry. [1] [5]

3.7 Modular arithmetic

There are an infinite number of integers and hence rationals and real numbers. Are there sets of numbers that behave somewhat like them but are finite?

Consider the integers modulo 6: $\{0, 1, 2, 3, 4, 5\}$. We write 3 mod 6 when we consider the 3 in this collection. Given any integer n, we can map it into this collection by computing the remainder modulo 6. Arithmetic can be done in the same way:

$$\begin{aligned} 7 &\equiv 1 \bmod 6 & \qquad (4-5) &\equiv 5 \bmod 6 \\ -2 &\equiv 4 \bmod 6 & (3\times 7) &\equiv 3 \bmod 6 \\ (5+4) \bmod 6 &\equiv 3 \bmod 6 & (2+4) &\equiv 0 \bmod 6 \end{aligned}$$

Instead of using "=", we write "≡" and say that a is *congruent* to b mod 6 when we see $a \equiv b \bmod 6$. This means that $a - b$ is evenly divisible by 6: $6|(a-b)$.

This is a group under addition with identity 0. In the last example, 2 is the additive inverse of 4. We denote this $\mathbb{Z}/6\mathbb{Z}$.

Question 3.7.1

What is -1 mod 6? For n a natural number greater than 1, what is -1 mod n?

Let's consider the same collection but without the 0. Instead of addition, use multiplication with identity 1.

Is this a group? Is it closed under multiplication? Does every element have an inverse?

Since $2 \times 3 = 6 \equiv 0$ and 0 is not in the collection, it is not closed under multiplication!

The elements that **do not** have multiplicative inverses are 2, 3, and 4 because each of these share a factor with 6.

The inverse of 1 is itself. $(5 \times 5) \bmod 6 \equiv 5^2 \bmod 6 \equiv 25 \bmod 6 \equiv 1 \bmod 6$, so it too is its own inverse.

If we restrict ourselves to the elements that do not have a factor in common with 6 then we do get a group $\{1, 5\}$, though it is not a very big one. The fancy mathematical way of writing this group is $(\mathbb{Z}/6\mathbb{Z})^\times$.

If instead of 6 we had chosen 15, then the elements in $(\mathbb{Z}/15\mathbb{Z})^\times$ would be

$$\{1, 2, 4, 7, 8, 11, 13, 14\}.$$

We have a better was of expressing "does not have a factor in common" and that is via the greatest common divisor. The integers a and b do not share a non-trivial factor if and only if $\gcd(a, b) = 1$. That is, a and b are coprime. If this is the case, there are integers n and m such that $an + bm = 1$. If we look at this modulo b then

$$1 \equiv an + bm \bmod b \equiv an \bmod b.$$

Hence n is equal to a^{-1} modulo b!

The phase "X if and only if Y" means "if X is true, then Y is true, and if Y is true, then X is true." We cannot have one of them true and the other false.

The integers a that are in $(\mathbb{Z}/15\mathbb{Z})^\times$ have $0 < a < 15$ and $\gcd(a, 15) = 1$. If we use 7 instead, the elements in $(\mathbb{Z}/7\mathbb{Z})^\times$ are $\{1, 2, 3, 4, 5, 6\}$.

Well this case is interesting! We got all the non-zero elements *because 7 is prime*. If p is a prime number then none of the numbers $1, 2, 3, \ldots, p - 1$ share a factor with p. That is, they are coprime with p.

The elements $1, 2, \ldots, p - 1$ form a multiplicative group with identity element 1 if and only if p is prime. The group has $p - 1$ elements.

The elements $0, 1, 2, \ldots, p - 1$ form a field under "+" and "×" with identity elements 0 and 1, respectively, if and only if p is prime. The field has p elements and is denoted $\mathbb{F}_p$.

There are more finite fields than these but any other finite field that is not formed this way is an extension of one of these. The number of elements in any finite field is a power of a prime number p. Any two finite fields with the same number of elements are isomorphic.