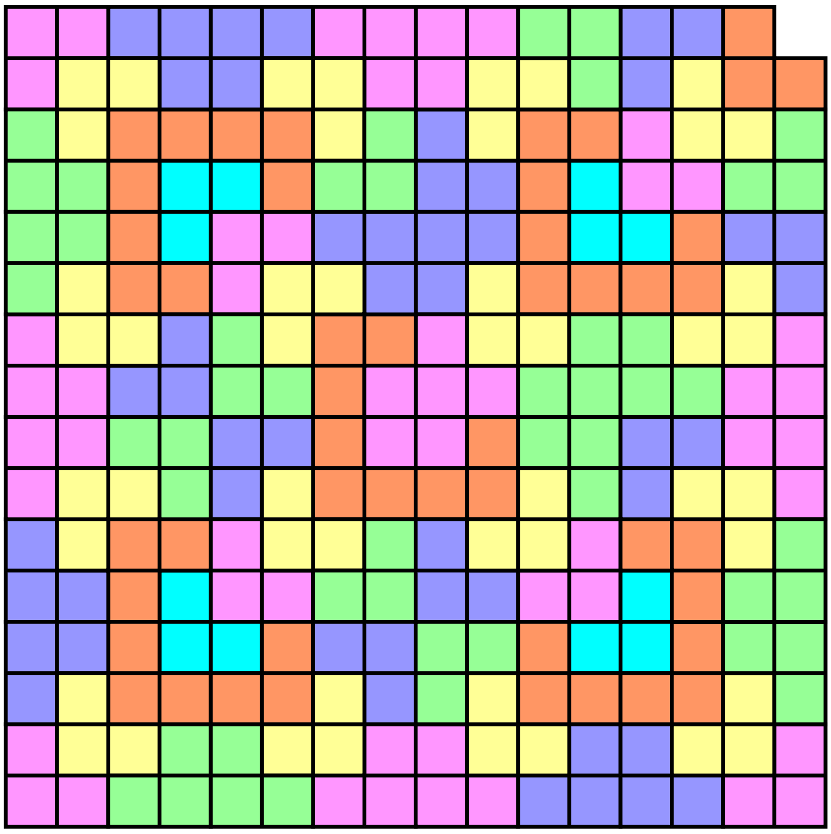
COMPSCI 120
Mathematics for Computer Science


Above: How to tile a 16×16 grid minus its top-right corner with shapes.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Contents

Compsci 120 University of Auckland

How To Read This Coursebook

2020 *Chapter -1*

Welcome to Compsci 120: Mathematics for Computer Science! We're excited to have you in this paper, and cannot wait to show you some of the fundamental mathematics that underpins the field of computer science.

This coursebook is possibly a bit different to ones you have seen in classes before. It has the following features:

- Each chapter opens up with a set of "motivating problems." These are exercises and puzzles related to the theme of each chapter, chosen to illustrate how the mathematical content covered in this paper can be used to solve "real-world" tasks.

  After reading these problems, try pausing for a bit and solving them on your own before going on to the rest of the chapter! Doing so will help you understand the concepts in each section better than immediately reading through everything. (It will also help you appreciate the solutions to these puzzles when they show up later in the chapter!)

- We've placed large margins on the side of each page, for you to write notes in. When you're in class or reading the book at home, use these margins to write down questions that come to mind, useful ideas from class, or as scratchwork when you're trying to work out an exercise.

A mathematical poem, to start off your margin notes:

A dozen, a gross, and a score,
plus three times the square root of four,
divided by seven,
plus five times eleven,
is nine squared and not a bit more!

- Finally, we've placed even more exercises at the end of each chapter. Easier/more straightforward problems are labeled (-), while trickier ones are labeled with a (+). Try working on these problems to test your understanding!

  Some exercises are labeled (++); these are exceptionally hard and/or open problems in mathematics. We're not expecting anyone to solve these problems; instead, we're listing them so that you can see the kinds of tasks that you might study in a Ph.D programme. (If you do solve any of these, though, *please* let us know!)

  While solutions to these end-of-chapter problems are not present in this coursebook, your lecturers are happy to give you hints or walk you through the solutions to any of these problems in office hours or on Piazza (a discussion forum you can find on Canvas!)

Have fun, and enjoy the course!

Compsci 120 University of Auckland

Expected Background Knowledge

2020 *Chapter 0*

0.1 What Does Compsci 120 Expect?

We are assuming that students entering this paper are relatively mathematically confident, having done well in NCEA Level 3 mathematics. To be precise: we're hoping that you've received a 'merit' or higher on one of the three externally-assessed NCEA L3 mathematics standards, namely differentiation, integration, or complex numbers.

In other systems, this is roughly equivalent to a passing mark in CIE A2 mathematics, or a C or better in CIE AS mathematics, or a 3/7 or higher in IB mathematics.

Compsci 120 is a paper that assumes you are already comfortable with a number of mathematical concepts and conventions. Trying to take this course without a solid background in mathematics is a **bad idea.** If you do not have this background, **you should take Maths 102!** Maths 102 is a course ran every semester (including summer semesters) at the University of Auckland, and has no prerequisites. It is designed to give you the skills to succeed in CS 120!

To help clarify some of the specific concepts we're expecting students to understand (in case you're coming from overseas, or it's been a while since you finished high school), here is a particular set of skills that we are hoping you've acquired over your career. This list is not exhaustive, but is just meant to point out the most common stumbling blocks that less-prepared students encounter in Compsci 120. Again, if you do not feel comfortable with these calculations, please enrol in Maths 102!

By "comfortable with," we mean that you should be able to see these calculations done in lecture without further explanation, and be able perform these calculations yourself on an exam without a calculator.

- **Exponents.** You should know what a^b is for any two integers a, b, and know how to work with exponents. For example, the following calculations are ones you should be comfortable with:

$$\begin{aligned} 2^5 &= 2\cdot 2\cdot 2\cdot 2\cdot 2 = 32, & 10^0 &= 1,\\ 2^3\cdot 2^4 &= 2^{3+4} = 2^7 = 128, & a^b\cdot a^c &= a^{b+c},\\ \left(2^2\right)^3 &= 2^{2\cdot 3} = 2^6 = 64, & \left(a^b\right)^c &= a^{bc},\\ 7^{-2} &= \frac{1}{7^2} = \frac{1}{49}, & a^{-b} &= \frac{1}{a^b}. \end{aligned}$$

- **Logarithms.** You should know what $\log_2(n)$ and $\log_{10}(n)$ mean, and be comfortable with calculations like the following:

$$\log_2(32) = \log_2\left(2^5\right) = 5, \qquad \log_{10}(1) = 0, \qquad \log_2(2^n) = n, \qquad 2^{\log_2(a)} = a.$$

- **Expanding polynomials.** You should be able to expand products of polynomials and other expressions. For example, the following calculations are ones that you should be able to read and understand without further explanation:

$$(x+1)^3 = x^3 + 3x^2 + 3x + 1,$$

$$\begin{aligned} (x-y)(x^n + x^{n-1}y + x^{n-2}y^2 + \ldots + xy^{n-1} + y^n) &= x^{n+1} \; + x^n y + x^{n-1}y^2 + \ldots + xy^n\\ &\qquad - x^n y - x^{n-1}y^2 - \ldots - xy^n - y^{n-1}\\ &= x^{n+1} \; - y^{n+1}, \end{aligned}$$

- **Fractions**. You should be able to combine ratios and fractions with various arithmetic operations. For instance, you should be comfortable with the following calculations:

$$\frac{1}{3} - \frac{4}{8} = \frac{1 \cdot 8 + (-4) \cdot 3}{24} = -\frac{4}{24}, \qquad \left(-\frac{45}{11}\right) \cdot \left(-\frac{5}{6}\right) = \frac{(-45) \cdot (-5)}{11 \cdot 6} = \frac{225}{66},$$

$$3 + \frac{7}{113} = \frac{113 \cdot 3 + 7}{113} = \frac{346}{113}, \qquad \frac{a}{b} + \frac{d}{bc} = \frac{ac + d}{bc},$$

$$\frac{1}{1-x} = \frac{1 + -x + x}{1-x} = 1 + \frac{x}{1-x}, \qquad \frac{1}{\sqrt{x} - a} = \frac{1}{\sqrt{x} - a} \cdot \frac{\sqrt{x} + a}{\sqrt{x} + a} = \frac{\sqrt{x} + a}{x - a^2}.$$

- **Solving equalities and inequalities.** Given an equation or inequality in one or more variables, you should be able to rearrange it to "solve" for one variable in terms of the others. For example, the following processes should be ones you're comfortable with:

$$3x - 4 = 12 \quad \Rightarrow \quad 3x = 16 \quad \Rightarrow \quad x = \frac{16}{3},$$

$$(a^2 - 1)b + 1 = a \quad \Rightarrow \quad (a^2 - 1)b = a - 1 \quad \Rightarrow \quad b = \frac{a-1}{a^2-1} = \frac{1}{a+1}, \quad \text{if } a \neq \pm 1$$

$$-5x - 7 \leq 3 \quad \Rightarrow \quad -5x \leq 10 \quad \Rightarrow \quad x \geq -2,$$

$$\frac{20}{x^2+1} > 2 \quad \Rightarrow \quad \frac{x^2+1}{20} < \frac{1}{2} \quad \Rightarrow \quad x^2 + 1 < 10 \quad \Rightarrow \quad |x| < 3,$$

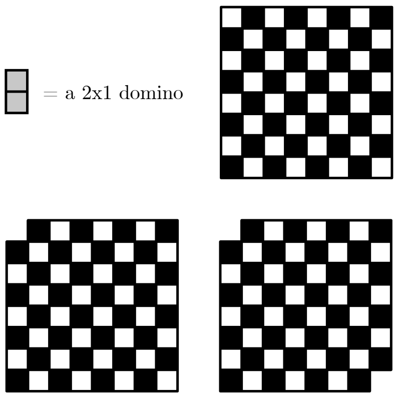
$$\begin{aligned} x^2 - 3x + 2 > 0 \quad \Rightarrow \quad (x-2)(x-1) > 0 \quad &\Rightarrow (x-2) \text{ and } (x-1) \text{ are both } < 0, \text{ or } (x-2) \text{ and } (x-1) \text{ are both } > 0 \\ &\Rightarrow (x < 2 \text{ and } x < 1), \text{ or } \quad (x > 2 \text{ and } x > 1) \\ &\Rightarrow (x < 1) \text{ or } (x > 2). \end{aligned}$$

- **Substitution**. Given a function $f(x)$, you should be able to plug in values and expressions in to this function, and get the correct output. For example, you should be capable of the following:

$$\text{If } f(x) = x^2 + 1 \quad \text{then} \quad f(4) = 4^2 + 1 = 17, \quad \text{and} \quad f(x+1) = (x+1)^2 + 1 = x^2 + 2x + 2.$$

$$\text{If } g(n) = \frac{n(n+1)}{2} \quad \text{then} \quad g(n+1) = \frac{(n+1)(n+2)}{2}, \quad \text{and} \quad g(2n) = \frac{(2n)(2n+1)}{2} = n(2n+1).$$

$$\text{If } h(x) = 1 - x \quad \text{then} \quad h(y) = 1 - y, \quad \text{and} \quad h(h(y)) = h(1-y) = 1 - (1-y) = y.$$



Exercise 1.1. *Take an 8×8 chessboard. Suppose you have a bunch of 2×1 dominoes lying around. Can you completely cover your chessboard with dominoes, so that the dominoes don't overlap or hang off of the board? Now, suppose you have a younger sibling who has eaten the top-left square of your chessboard. Can you completely cover without overlap this chessboard with 2×1 dominoes?*

What if they also ate the bottom-right square? Does this change your answer?

Exercise 1.2. *A mistake people often make when adding fractions is the following:*

$$\frac{1}{x} + \frac{1}{y} \overset{?}{=} \frac{1}{x+y}$$

Now, we know that this isn't right, and that $\frac{1}{x} + \frac{1}{y}$ is actually $\frac{y}{xy} + \frac{x}{xy} = \frac{x+y}{xy}$. Sometimes, however, even a formula that is wrong in general might be right in a specific situation! (Think of the old adage "a stopped clock is right twice a day.")

Does this ever happen with this mistake? In other words: are there any values of x, y such that $\frac{1}{x} + \frac{1}{y} = \frac{1}{x+y}$? Or is this false for literally every value of x and y?

1.1 Integers

We start our coursebook by studying the **integers**! In case you have not seen the word "integer" before, we define it here:

In this course and in mathematics in general, we will define lots of useful concepts. When we do so, we'll label these things by writing "**Definition**" in bold, and then give you a carefully-written and precise definition of that word.
When studying for this class, it is a good idea to make sure that you know all of the definitions in this coursebook, as well as some examples and nonexamples for each definition where appropriate.

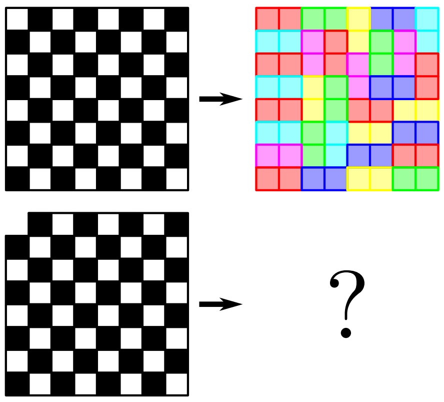
Definition 1.1. *The **integers** are the collection of all whole numbers: that is, they consist of the whole positive numbers $1, 2, 3, 4, \ldots$, together with the whole negative numbers $-1, -2, -3, -4, \ldots$, and the number 0. We denote this set by writing the symbol $\mathbb{Z}$.*

The symbol $\mathbb{Z}$ comes from the German word "Zahl," which means "number," in case you were curious.

Example 1.1. The numbers $1, -7, 10000, 78, 45, 0, -345678$ are integers, but things like $\sqrt{2}, -2.787878787, \frac{1}{8}$ and π are not.

In some form or another, integers have been used by humans for almost as long as humans have existed theirselves. The Lebombo bone, one of the oldest human artifacts, is a device on which people used tally marks to count the number of days in the lunar cycle. The base-10 system and idea of numerals took a bit longer for us to discover, but we can trace these concepts back to at least the Egyptians and Sumerians around 4,000-3,000 BC. Finally, negative numbers and zero have been with us for at least two millenia: historical documents tracking back to at least 1000 BC describe how Chinese mathematicians used negative numbers solve concrete problems on bounding areas of fields, exchanging commodities, and debt.

In short, integers are quite handy! To this day, we use them to model all sorts of problems. To give a pair of examples, let's solve the first two exercises from this chapter:



Answer to Exercise 1.1. By just trying things out by hand, it's relatively straightforward to find a tiling of a 8×8 board with no squares missing; one solution is drawn in the margin. When you remove a square, however, things get a bit more interesting! Try it for a while; no matter how you do this, you'll always have at least one square left over uncovered.

However, saying "I tried it a lot and it didn't work" is not a very persuasive argument. Suppose that you had a boss that said that Chad from Marketing said this was totally possible, and to have a working version on their desk by 5pm or you'll be fired. What do you do?

Well, maybe you start revising your resumé and looking for other work. Before that, though, you may want to try to make a logical argument to your boss that shows that it's **impossible** to come up with such a tiling — a set of reasons so airtight that no matter what they think or would respond with, they'd have to agree that you're right. In mathematics and computer science, we call such logical arguments **proofs**!

In this class, we're going to write these kinds of arguments for almost all of the claims that we make. We're going to leave the rules for what constitutes a "proof" a little vague at first; in our first few chapters, we're going to just try to write a solid argument that tells us why something is true, and anything that does that we'll count as a proof. (Later in this course, we'll approach proofs a bit more formally: feel free to read ahead if you can't wait!)

For this problem, let's **prove** that an 8×8 chessboard without a single square cannot be covered with 2×1 dominoes as follows:

Proof. First, notice that each time we place a 2×1 domino on a board, it covers **two** squares worth of area. Because dominoes cannot overlap, this means that if we have placed k dominoes on our board, there are $2k$ squares of our board covered by dominoes.

If we have an 8×8 chessboard that's missing a square, that board has $8 \cdot 8 - 1 = 63$ squares on it in total. Therefore, if we've completely covered our chessboard with dominoes, the number k of dominoes we used must satisfy the equation $2k = 63$.

In other words, we have $k = 31.5$; i.e. we've used half of a domino! This means that the other half of that domino is sticking off of our board or overlaps another domino, which means we've broken the rules of our tiling problem. So this is impossible! □

To denote the end of our arguments, we draw a square box symbol, like □. This means that we've reached the end of our argument. In physics or other fields, people often use QED in place of this. Feel free to use either in your own working!

Notice how in the argument above, the only things we used were factual observations (i.e. the number of squares in our 8×8 grid minus a square, the amount of area taken up by each 2×1 domino) and logical combinations of those observations. This is how you write a solid argument!

To get a bit more practice with this sort of thing, let's try the last part of our exercise. In this problem, we had a chessboard with its top-left **and** bottom-right squares removed, and wanted to try to cover this with 2×1 dominoes.

On one hand, the argument above no longer tells us that this is impossible. A 8×8 board minus two squares contains 62 squares in total; therefore, it might be possible to do this with 31 dominoes! However, if you try this for a while, you'll find yourself quite stuck again. So: *why* are we stuck? What is a compelling argument we could write that would persuade someone that this is truly impossible, and not just that we're bad at tiling?

After some thought and clever observations, you might come up with the following:

Proof.

i. Notice that each time we place a 2×1 domino on our grid, it's not just true that it covers two units of area: it **also** covers exactly one unit of white area and one unit of black area! This is because our chessboard has alternating white and black squares.
ii. Because dominoes cannot overlap, this means that if we have placed k dominoes on our board, there are k white squares and k black squares covered by our dominoes. In particular, this means that we always have as much black area covered as white area!
iii. However, if we count the black and white area in our 8×8 board with the top-left and bottom-right squares removed, we can see that there are 32 black squares and 30 white squares. These numbers are different! Therefore, it is impossible to cover this board with dominoes as well.

□

Notice that when we wrote this argument, we used observations from our earlier work, and modified them to suit this new problem! This is a good problem-solving technique in general: always see if you can modify a pre-existing solution before trying to come up with something entirely new.

In our working above, the fact that we could not write 63 as an integer multiple of 2 was a very useful observation! We can generalize this into the concepts of **even** and **odd** integers:

1.2 Even and Odd Integers

Definition 1.2. *We say that an integer is* ***even*** *if we can write it as 2 times another integer; in other words, we say that an integer n is even if we can find an integer k such that $n = 2k$.*

Similarly, we say that an integer is ***odd*** *if we can write it as one plus an even number; in other words; we say that an integer n is odd if we can find an integer k such that $n = 2k + 1$.*

Example 1.2. 2 is even, because we can write $2 = 2 \cdot 1$, and 1 is an integer. Similarly, 4 is an even number, because we can write $4 = 2 \cdot 2$ and 2 is an integer.

As well, $-2, -4$ and -6 are all even, because we can write $-2 = 2 \cdot (-1)$, $-4 = 2 \cdot (-2)$ and $-6 = 2 \cdot (-3)$, where $-1, -2, -3$ are all integers.

3 is odd, because we can write $3 = 2 + 1$ and 2 is even (as shown above.) Similarly, because $5 = 4 + 1$ and $7 = 6 + 1$, we have that 5 and 7 are also odd. As well, $-1, -3, -5$ are all odd, as $-1 = -2 + 1, -3 = -4 + 1$, and $-5 = -6 + 1$.

Exercise 1.3. *Using the definition above, is 0 even or odd?*

To get some practice writing logical arguments, let's look at a few properties of even and odd numbers that you already know:

Claim 1.1. *The sum of any two odd numbers is even.*

Before getting into a "good" solution for this claim, let's first study an argument that **doesn't** work.

"Bad" solution: Well, $1+1=2$ is even, $3+7=10$ is even, $-13+5=-8$ is even, and $1001+2077=3078$ is even. Certainly seems to be true!

□

A defense of the "bad" solution: This might seem like a silly argument, but suppose we'd listed a thousand examples, or a billion examples, or set a computer program to work overnight and had it check all of the pairs of numbers below 10^{12}. In many other fields of study, that would be enough to "show" a claim is true! (Think about science labs: there, we prove claims via experimentation, and any theory you could test a billion times and get the same result would certainly seem very true!)

Why this argument is not acceptable in mathematics and computer science: When we make a claim about "any" number, or say that something is true for "all" values, we want to **really** mean it. If we have not literally shown that the claim holds for every possible case, we don't believe that this is sufficient!

This is not just because computer scientists are fussy. In the world of numbers, there are tons of "eventual" counterexamples out there:

- Consider the following claim: "The sequence of numbers "12, 121, 1211, 12111, 121111, …" are all not prime."

  If you were to just go through and check things by hand, you'd probably be persuaded by the first few entries: $12 = 3\cdot4, 121 = 11\cdot11, 1211 = 173\cdot7, 12111 = 367\cdot11, 12111 = 431\cdot281, \ldots$

  Skip ahead a bit to definition 1.4 if you haven't encountered prime numbers before!

  However, when you get to $12\overbrace{111\ldots1}^{136\ 1's}$, that one's prime! This is well beyond the range of any reasonable human's ability to calculate things, and yet something that could come up in the context of computer programming and information security (where we make heavy use of 500+ digit primes all the time.)

- Here's a fun exercise one of the writers of this coursebook saw in the wild on Facebook, shared by one of our troll-ier friends:

  Exercise 1.4. *(++) Can you find positive integer values for* 🐐, 🐤 *and* 🐍 *so that the equation* $\frac{🐍}{🐐+🐤} + \frac{🐐}{🐍+🐤} + \frac{🐤}{🐍+🐐} = 4$ *holds?*

  On its surface, this looks simple, right? Just solve it.

  And yet: suppose you set a computer program to work at this problem, by just bashing out all possible triples of numbers. To be fair, let's give you access to the world's fastest supercomputer. By the end of a week, you wouldn't have found an answer. By the end of a year, you wouldn't have found an answer! Indeed: by the time we reached the heat death of the universe, you'd still have found nothing. You'd be tempted to say that no answer exists, right?

  Roughly 10^{106} years, or $3\cdot10^{113}$ seconds, or about $6\cdot10^{130}$ floating-point operations on the world's fastest supercomputer as of early 2019.

  And yet, an answer exists! The math required to find this goes *way* beyond the scope of this course, but if you're curious: the smallest known answer is to do the following:

🐍=154476802108746166441951315019919837485664325669565431700026634898253202035277999,

🐐= 36875131794129999827197811565225474825492979968971970996283137471637224634055579,
and

🐤= 4373612677928697257861252602371390152816537558161613618621437993378423467772036.

In general, mathematics and computer science is full of things like this! Huge numbers that are prime or that satisfy equations similar to the ones above can be incredibly useful in information security (as you'll see in later papers in Compsci/Maths!)

Examples alone, then, are not enough for a good argument. What *would* a good solution look like here? Here is one of many possible answers:

Claim 1.1. *The sum of any two odd numbers is even.*

Proof. Take any two odd numbers. Let's give them names, for ease of reference: let's call them M and N. By definition, because M and N are odd, we can write $M = 2k + 1$ and $N = 2l + 1$, for two integers k, l.

Therefore, $M + N = (2k + 1) + (2l + 1) = 2k + 2l + 2 = 2(k + l + 1)$. In particular, this means that $M + N$ is an even number, as we've written it as a multiple of 2! $\square$

The argument above might look a little awkwardly-written to you at a first glance! This is because mathematics is something of a language in its own right: there are certain phrases and constructions with very specific meanings in mathematics, that we use when making arguments to ensure that they're completely rigorous. Let's talk about some of those phrases and concepts that came up in the argument above:

- We worked in *general*! That is, we didn't just look at a few examples, but instead considered arbitrary values! This is what writing a phrase like "Take any two odd numbers" does for us: it doesn't lock us into specific odd numbers, but instead tells the reader that we're going to show that this works for anything that could ever come up.
- We defined our variables! That is, we didn't just try to use pronouns to refer to our odd numbers the whole way through: instead, we gave them variable names, by calling them N and M. This makes writing our arguments much cleaner, as it's much easier to refer to something if it has a name!

  Note also that we defined these variables only **after** saying what kind of object they were! This is like when you're writing code: you typically have to declare the type of variable you're working with, i.e. you'd write something like "`int numA`" instead of just saying "`numA`."
- We used words to describe what we were doing and why it worked! Mathematics, surprisingly, has a lot of words and sentences in it. You should find that the number of sentences in most solutions you make in this class exceeds the number of equations!

To get a bit more practice with this, let's try out a few more claims:

Claim 1.2. *The product of any two odd numbers is odd.*

Proof. As before, let's start by taking any two odd numbers. Let's give them names, and call them M and N respectively.

Also as before, let's consult our definitions! By definition, because M, N are both odd, we can again write $M = 2k + 1$ and $N = 2l + 1$, for two integers k, l.

Therefore, $M \cdot N = (2k + 1) \cdot (2l + 1)$. Expanding this product gives you $4kl + 2k + 2l + 1$, which you can regroup as $(4kl + 2k + 2l) + 1$. Factoring a 2 out of the left-hand part gives you $2(2kl + k + l) + 1$.

Because k, l are integers, the expression $2kl + k + l$ inside the parentheses is an integer as well, as any product or sum of integers is still an integer.

Therefore, we've written $M \cdot N$ in the form $2 \cdot (\textit{an integer}) + 1$; i.e. we've shown that $M \cdot N$ satisfies the definition of being an odd integer! □

Claim 1.3. *No integer is both even and odd at the same time.*

Proof. As before, let's start by working in general. Take any integer N.

As a thought experiment, let's think about what it would mean for N to be both odd and even at the same time. If N was even, then by definition we could write $N = 2k$ for some integer k; as well, if N is also odd, then by definition we should be able to write $N = 2l + 1$ for some integer l.

Phrases like "Take any integer n" or "take any number x" are useful; they start by both saying that we're going to work in general with any number, and also assign a variable to that number so that we can refer to it.

Note that we had to pick different letters k, l when we applied the even and odd definitions! If we had used the same letter k for both, that would imply that the same integer is being used in both definitions, and this might not be true.

As a result, we have $N = 2k$ and $N = 2l+1$. Combining gives us $2k = 2l+1$, which we can rearrange into $2(k - l) = 1$; i.e. $k - l = \frac{1}{2}$.

But this is clearly not possible! k and l are both integers, and so their difference must an integer as well; it cannot be $\frac{1}{2}$!

As a result, we've shown that it is impossible for an integer N to be equal to $2k$ and $2l + 1$ at the same time if k, l are both integers; that is, it is impossible for N to be both odd and even! □

By definition, you can think of our even/odd split above as classifying every number as either "a multiple by 2" or "not a multiple of 2." We expand on this idea in the following section, where we study **divisibility** and **prime numbers**.

1.3 Divisibility and Primes

Definition 1.3. *Given two integers a, b, we say that a **divides** b if there is some integer k such that $ak = b$.*

There are many synonyms for "divides": each of the phrases

- *"a is a divisor of b",*
- *"a is a factor of b",*
- *"b is a multiple of a",*
- *"b can be divided by a," and*
- $a \mid b$

all mean the same thing as "a divides b."

Here's a string of examples, to make this clearer:

Example 1.3.

- 4 divides 12; this is because we can multiply 4 by 3 to get 12.
- 72 can be divided by -6; this is because we can multiply -6 by -12 to get 72.
- 2 does not divide 15; this is because for any integer k, $2k$ is an even number, and so is in particular never equal to an odd integer like 15.
- n is a multiple of 1 for any integer n; this is because we can always multiply 1 by n to get n.
- n is a factor of 0 for any integer n; this is because we can always multiply n by 0 to get 0.

Example 1.4. Note that the phrases "divides" and "can be divided by," while quite similar-sounding in English, have almost the opposite meaning in mathematics! For instance, **3 divides 9** and **9 can be divided by 3** are the same claims.

To get some practice with making abstract arguments about divisibility, let's study a simple claim about factors and divisibility:

Claim 1.4. *Let a, b, c be three integers. If a divides b and b divides c, then a divides c.*

Proof. Again, we need to work in abstract! That is: we cannot just check this for a few values and say that "well, when $a = 4, b = 12, c = 36$ this all works out." Instead, we must consider **any** three integers a, b, c, and work in general without knowing what a, b, c are.

In theoretical computer science / mathematics, the words "any" and "every" are often used interchangeably. That is, if someone says to you "For any integer n, $n^2 \geq 0$," this means the same thing as "For every integer n, $n^2 \geq 0$."

By definition, if a divides b, we can write $ak = b$ for some integer k. Similarly, if b divides c, then we can write $bl = c$ for some integer l.

Now, take the equation $ak = b$, and use this to substitute in ak for b in our second equation $bl = c$. This gives us $akl = c$, i.e. $a(kl) = c$. Because k, l are both integers, their product is an integer; as a result, we've written $a \cdot (\textit{an integer}) = c$. In other words, by definition we have shown that a is a factor of c, as desired. $\square$

A particularly useful concept related to divisibility is the concept of a **prime number**:

Definition 1.4. *A* ***prime number*** *is any positive integer with only two distinct positive factors; namely, 1 and itself.*

Example 1.5. The first few prime numbers are $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots$

Observation 1.1. *1 is not a prime number.*

Proof. This is because 1's only factor is 1, and so 1 does not have two *distinct* positive integer factors. $\square$

Observation 1.2. *2 is the only even prime number.*

Proof. This is because every other positive even number by definition has the form $2k$, and so has at least $1, 2, k, 2k$ as its set of factors (and thus has more than 2 distinct positive factors.) $\square$

Definition 1.5. *A* ***composite number*** *is any positive integer n that can be written as the product of two integers a, b, both of which are at least 2 (and thus both of which are strictly smaller than n.)*

Example 1.6. $6 = 2 \cdot 3$, $9 = 3 \cdot 3$, and $24 = 2 \cdot 12$ are all composite.

Observation 1.3. *By definition, any positive integer is either a prime number, a composite number, or 1.*

Definition 1.6. *Given a positive integer n, a* ***prime factorization*** *of n is any way to write n as a product of prime numbers.*

Example 1.7. Here are a few prime factorizations:

- $120 = 2^3 \cdot 3 \cdot 5$,
- $243 = 3^5$,
- $30031 = 59 \cdot 509$

Here are a pair of useful observations about prime numbers:

Theorem 1.1. *Every positive integer can be factorized into a product of prime numbers in exactly one way, up to the ordering of those prime factors.*

To see a proof of Theorem 1.1, take Compsci 225!

You might object to Theorem 1.1 by saying that 1 cannot be factored into a product of prime numbers. If so, good thinking! For now, regard 1 as a special case.

Later on, though, we'll consider the idea of an "empty product" when we get to the factorial and exponential functions. At that time, we'll argue that the "empty product" or "product of no numbers" should be considered to be 1, because 1 is the multiplicative identity. If you believe this, then 1 can indeed be written as the product of prime numbers; it specifically can be written as the product of **no** prime numbers!

In other words, this theorem is saying the following:

- Every positive integer can be factored into primes, as illustrated above. So, for example, we can take 60 and write it as $2 \cdot 2 \cdot 3 \cdot 5$.
- No number can be factored into primes in two different ways, up to the ordering. That is: while you could write 60 as $5 \cdot 2 \cdot 3 \cdot 2$ or $5 \cdot 3 \cdot 2 \cdot 2$, you're never going to write a prime factorization of 60 that has a 7 as one of its prime factors, or doesn't have a 5.

Proving this theorem is a bit beyond our skill set at the moment. Instead, let's mention a second useful fact about primes:

Theorem 1.2. *There are infinitely many primes.*

This proof is also a bit beyond us for now. However, if you skip ahead to the proof by contradiction section of our notes / wait a few weeks, you'll see a proof of this in our course!

Prime numbers (as you'll see in Compsci 110) are incredibly useful for communicating securely. Using processes like the RSA cryptosystem, one of the first public-key cryptosystems to be developed, you can use prime numbers to communicate secretly over public channels.

Prime numbers are *also* quite baffling objects, in that despite having studied them since at least 300 BC there are still so many things we do not know about them! Here are a few particularly outstanding problems, the solutions for which would earn you an instant Ph.D/professorship basically anywhere you like in the world:

Exercise 1.5. *(++)(Goldbach conjecture.) Show that every even integer greater than 2 can be written as the sum of two prime numbers. For example, we can write* $8 = 3 + 5, 14 = 11 + 3, 24 = 17 + 7, 6 = 3 + 3, \ldots$

Exercise 1.6. *(++)(Twin prime conjecture.) A pair of prime numbers are called* ***twin primes*** *if one is exactly two larger than the other. For example,* $(5, 7)$, $(11, 13)$ *and* $(41, 43)$ *are twin primes. Show that there are infinitely many twin primes.*

In general, working with prime numbers can get tricky very fast; even simple problems can get out of hand! With that said, there are some claims that are approachable. We study two such statements here:

Claim 1.5. *Let ab be a two-digit positive integer (where b is that number's ones' digit and a is its tens' digit.) Show that the number $abab$ is not prime.*

Proof. As noted before, examples alone aren't enough for a solution. However, if you're stuck (as many of us would be on first seeing a problem like this,) they *can* be useful for helping us find a place to start!

ab	$abab$	prime factorization of $abab$
10	1010	$2 \cdot 5 \cdot 101$
98	9898	$2 \cdot 7^2 \cdot 101$
21	2121	$3 \cdot 7 \cdot 101$
88	8888	$2^3 \cdot 11 \cdot 101$
92	9292	$2^2 \cdot 23 \cdot 101$
43	4343	$43 \cdot 101$

So: let's create a bunch of two-digit numbers and factor them (say, via WolframAlpha.) If our claim is wrong, then maybe we'll stumble across a number whose only factors are 1 and itself. Conversely, if our claim is right, maybe we'll see a pattern we can generalize! We do this at right.

As we do this, a pattern quickly emerges: it looks like all of these numbers are multiples of 101! This isn't a solution yet: we just checked six numbers out of quite a few possibilities. It does, however, tell us how to write a proper argument here:

Notice that for **any** two-digit number ab, $ab \cdot 101 = ab \cdot 100 + ab = ab00 + ab = abab$. Therefore, any number of the form $abab$ is a multiple of 101 and also a multiple of ab, by definition. Because ab is a two-digit number by definition, this means that $abab$ has at least two factors other than 1. This means that $abab$ cannot be a prime number, as claimed! $\square$

Claim 1.6. *Let n be any positive integer greater than 1. Let $k = \lfloor \sqrt{n} \rfloor$ denote the number given by rounding down $\sqrt{n}$. If n does not have any of the numbers $2, 3, \ldots k$ as factors, then n is prime.*

Proof. We use the "suppose we're wrong" technique from Claim 1.3. That is: take any number n with the "no factors in the set $2, 3, \ldots k$" property listed above. There are two possibilities:

- n is prime. This is what we want: if this case holds, we're done!
- n is not prime. We want to show that this case cannot hold; if we can do this, then we are left with only the case above, and have thus proven our claim!

  To do this: note that if n is not prime, then, it must have more than 2 factors. Let a be one of those positive factors that is not 1 or n; by the definition of factor, then, we can write $n = ab$ for some positive integer b, and thus have that b is also another factor of n.

  We know that because $a \neq n$ that $b \neq 1$. We also know that because both a, b are factors of n, that by our "suppose we're wrong" assumption that $a, b \neq 2, 3, 4, \ldots k$. Therefore, by combining these results, both $a, b > k$; that is, both a, b are greater than $\sqrt{n}$.

  But this means that $ab > \sqrt{n} \cdot \sqrt{n} = n$. But this is impossible, as $ab = n$.

Therefore the only possibility left is that n is prime, as desired. $\square$

This last claim is particularly useful! We know that by definition, a positive integer n is prime if it has exactly two positive integer factors: 1 and itself. Naively, then, to check if a number n is a prime, you might think that you would have to check all of the numbers $2, 3, 4, \ldots n - 1$ each individually, and see if any of them divide n.

However, Claim 1.6 tells us that we don't actually have to check all the numbers up to $n - 1$; we only have to go up to $\sqrt{n}$. This can save us a lot of time: as you'll see later in this coursebook in our runtime-analysis section / in papers like Compsci 130 and 220, a "runtime" of checking $\sqrt{n}$ cases is considerably better than a "runtime" of checking n cases!

For instance, to see if $n = 101$ is prime, Claim 1.6 tells us that because 101 is not a multiple of 2,3,4,5,6,7,8,9 or 10, then 101 is itself a prime. (Much faster than checking *all* of the numbers up to 100.)

We leave a few more exercises along these lines for the end of this chapter. For now, we turn to an extension of the ideas behind divisibility: **modular arithmetic**!

1.4 Modular Arithmetic

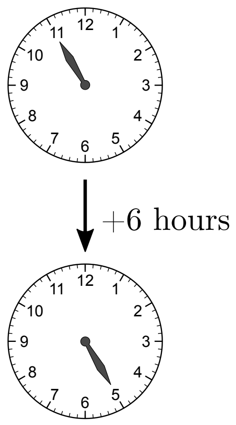
Modular arithmetic is something you've been working with since you were a child! Specifically, think about how you tell and measure **time**:

- Suppose that it is currently 6am and 3 hours pass. Because $6+3 = 9$, we know that the answer is 9am.

  Now, suppose that it is 11am and 18 hours pass. What is the time now?

Unlike the process above, you would not find the answer here by adding 18 to 11 and getting "29 o'clock." Instead, you'd find 11 + 18 = 29, and then **subtract off** 24 to get the right answer, $\boxed{\text{5 a.m.}}$

In general, when you're adding hours together to tell time, you do so on a circle (as drawn at right,) where measurements "wrap around" every 12 hours. That is: if it is 7 o'clock now and 38 hours pass, then the hour hand on a clock would point to a 9; this is because $7 + 36 = 45$, and $45 - 12 - 12 - 12 = 9$.



- In the above example, we saw that hours wrapped around in units of 12. Similarly, we know that days of the week repeat in groups of 7. That is: let Monday = 0, Tuesday = 1, and so on/so forth until we get to Sunday = 6. Then, suppose that it is Thursday and we want to know what day of the week it is in 10 days. We can see that because "Thursday +10" = $3 + 10 = 13$ and $13 - 7 = 6$, the day must be Sunday.
- Similarly, seconds wrap around in groups of 60. That is: if you type in "99" on your microwave and walk away, the same amount of time passes as if you had entered "1:39", because $99 - 60 = 39$ and thus 99 seconds is a minute and 39 seconds.

Using "99" on your microwave: one of the less-useful "life pro tips."

All of these calculations involved arithmetic where our numbers "wrapped around" once they got past a certain point. This idea is the basis for **modular arithmetic**, which is arguably the most useful mathematical concept you'll encounter in computer science.

To make this rigorous, let's make a definition:

Definition 1.7. *Take any two integers a, n, where $n > 0$. We define the number $a \% n$, pronounced "a mod n", by the following* ***algorithm*** *:*

- *If $a \geq n$, repeatedly* ***subtract*** *n from a until $a < n$. The result of this process is $a \% n$.*
- *If $a < 0$, repeatedly* ***add*** *n to a until $a \geq 0$. The result of this process is $a \% n$.*
- *If neither of these cases apply, then by definition $0 \leq a < n$. In this case, $a \% n$ is simply a (that is, we don't have to do anything!)*

We call % the ***modulus operator****. Most programming languages implement it as described here, though (as always) you should read the documentation for details.*

An **algorithm** is a step-by-step process for solving a problem or performing a calculation! We will study algorithms in more depth later in this class; you'll also see them everywhere in Compsci 110 / 130 / 220 / 225 / essentially, every paper you'll see in your Compsci degree!

Note: if $n < 0$, we can use the same process as above to calculate $a \% n$. The only change is that we replace n with $|n|$ in our steps, to compensate for the fact that n is negative.

To get a handle on this, we calculate a number of examples:

Example 1.8.
- $6 \% 2 = 0$. To see why, simply run the process above. Because $4 > 2$, we just subtract copies of 2 from 4 until we get something less than 2: $6 \xrightarrow{\text{subtract 2}} 4 \xrightarrow{\text{subtract 2}} 2 \xrightarrow{\text{subtract 2}} \boxed{0}$.
- $-9 \% 2 = 1$. To see why, simply run the process above. Because $-9 < 0$, we just add copies of 2 from 4 until we get something nonnegative: $-9 \xrightarrow{\text{add 2}} -7 \xrightarrow{\text{add 2}} -5 \xrightarrow{\text{add 2}} -3 \xrightarrow{\text{add 2}} -1 \xrightarrow{\text{add 2}} \boxed{1}$.
- In general, if n is any even number, then $n \% 2 = 0$. This is because if we start with any even number $n = 2k$, the process above will reduce $2k$ to 0 after subtracting/adding 2 k times in a row.
- Similarly, if n is any odd number, then $n \% 2 = 1$. These two observations come in handy when working with binary arithmetic: as you'll see in Compsci 110, these facts tell you that you can tell if a number is even or odd by looking at its last digit in binary!
- $3 \% 4 = 3$. This is an easy one to calculate: because $0 \leq 3 < 4$, we don't have to do anything here!

We also study a quick argument-based problem, to make sure that we're comfortable with a more "abstract" way of working with the modulus operator $\%$:

Claim 1.7. *If a, n are any two integers with $n > 0$, the quantity $a \% n$ exists and is between 0 and $n-1$. That is: the algorithm given above to calculate $\%$ will never "crash" nor "run forever," and it will always generate an output between 0 and $n-1$.*

Proof. As always, because this is a claim about **any** two integers, we need to work in general. That is: we cannot pick examples for a and n, and instead need to consider every possible pair of values for a, n with $n > 0$.

Our algorithm had three different cases: one for when $a \geq n$, one for when $a < 0$, and one for when $0 \leq a < n$. As such, our argument will likely want three cases as well:

- $a \geq n$. In this case, our algorithm repeatedly subtracted copies of n from a until it got to something less than n.

  This process clearly only runs for finitely many steps, as a starts off as greater than n and decreases by a fixed nonzero amount at each step. In particular, it must eventually be less than n, as that was the condition we gave for this process to stop.

  As well, if $a \geq n$, then $a - n \geq 0$; as a result, when this process ends, it cannot end at a negative number.

  Therefore, at the end of our process we've reduced a so that it's nonnegative and less than n, as desired.

- $a < 0$. In this case, our algorithm repeatedly added copies of n from a until it got to something positive.

  As before, this process clearly only runs for finitely many steps; this is because a starts off as a negative number and increases by a fixed amount at each step. In particular, it must eventually be nonnegative, as that was the condition we gave for this process to stop.

  As well, if $a < 0$, then $a + n < n$; as a result, when this process ends, it cannot end at a value equal to or greater than n.

  Therefore, at the end of our process we've reduced a so that it's nonnegative and less than n, again as desired.

- $0 \leq a < n$. In this case our algorithm just outputs a, which again has the desired properties.

Because every integer a falls into one of these three cases, we've proven our claim for all values of a and all $n > 0$, as desired. □

This sort of argument (that a given algorithm "works") is one that we will make repeatedly in this class! More generally, these are the most common sorts of arguments you'll want to make in computer science: when working with any problems that require nontrivial algorithms or thought, you'll often want to be able to write proofs that the process should be bug-free in theory.

Some useful notation for rounding: given any number x, we say that $\lfloor x \rfloor$ is "x rounded down," $\lceil x \rceil$ is "x rounded up," and $[x]$ is "x rounded to the nearest integer, with .5 rounding up." For example, $\lfloor \pi \rfloor = 3$, $\lceil \pi \rceil = 4$, $[\pi] = 3$, and $[3.5] = 4$.

Modular arithmetic comes up often in computer science and mathematics:

- **Remainders**. Take any two positive integers a, b. We say that the **quotient** of a on division by b is just "$\frac{a}{b}$ rounded down:" that is, we say that the quotient of $\frac{14}{3}$ is 4, as $\frac{14}{3} = 4.66666\ldots$ rounds down to 4.

Based on this, we say that the **remainder** of $\frac{a}{b}$ is the difference of a and the quotient of a on division by b times b; that is, the remainder of $\frac{a}{b}$ is everything "left over" after we try to divide through by b.

Notice that by definition, $a \,\%\, b$ is the remainder of $\frac{a}{b}$! So, for example, we have the following:

  - Because $14 \,\%\, 12 = 2$, we have that the remainder of $\frac{14}{12}$ is 2.
  - Because $26 \,\%\, 13 = 0$, we have that the remainder of $\frac{26}{13}$ is 0.

- **Binary arithmetic**. Computers are built off of the binary number system: i.e. instead of storing numbers in decimal notation where we use the digits 0,1,2,. . . 9, computers store everything with just 0's and 1's (i.e. on and off, which computers are much better at storing than something as imprecise as a decimal digit!) To work with numbers in binary, we often work modulo 2, as you'll see in classes like Compsci 110.

- **Overflow errors.** In many programming languages, you have to tell the computer the "type" of any variable that you want to work with. That is, you can't just tell the computer that "`x = 3`;" instead, you have to first tell the computer that x is an integer by writing something like "`int x;`" and then say "`x = 3`."

  However, when you declare that x is an integer, your computer does not automatically assume that x can literally be any integer! This is because when a computer declares a variable, it typically has to set aside a block of space to store the information corresponding to that variable. As a result, the computer has to make an assumption about a reasonable range of values that your integer will fall between (a common range is -2^{31} to $2^{31}-1$, i.e. you use 32 bits to describe your integer in binary, with one of those used to record $\pm$.)

  Suppose you did this, though, and later on tried to increase the value stored in x past its maximum value: i.e. you had $x = 2^{31}-1$, and you tried to add one to x. The result (if you ignore warnings / errors produced by your compiler) won't be 2^{31}, because this value is too large to store! Instead, it will often "overflow" and wrap around to become -2^{31} instead. That is: computers technically do a lot of their arithmetic "mod 2^{32}," and this can trip you up if you're not being careful about the sizes of your variables! Check out Wikipedia's integer overflow page for a bunch of examples and discussion around how to handle this situation.

- **Checksums**. On almost every ID number or tag (i.e. barcodes, bank account numbers, library book numbers, etc) the last digit or two is a "checksum" digit, found by adding up the previous numbers and applying the $\% n$ operator to the result (with n usually being 10, but sometimes something stranger like 97 in the case of bank account numbers.)

  The use of these checksum digits is that they let a computer program spot typos: if someone makes a mistake when entering a number, they'll usually do so in a way that changes the modulus of the sum! As a result, we can spot this error by rechecking the modulus, and alert the user that they've made a mistake.

- **Cryptography.** Calculating remainders modulo n has been key to the entire field of cryptography, from its inception in ancient Rome to its present-day uses. There is almost no way to keep information secret in the modern world that does not use modular arithmetic in some way.

As an example, let's look at how modular arithmetic is used in one of the first forms of cryptography: the **Caesar cipher**, otherwise known as a **shift** cipher!

a	b	c	d	e	f	g	h	i	j	k	l	m
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12
n	o	p	q	r	s	t	u	v	w	x	y	z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
13	14	15	16	17	18	19	20	21	22	23	24	25

– To set this up, take the alphabet $\{a, b, \ldots z\}$, and label each letter with the numbers $\{0, 1, \ldots 25\}$, as shown at right. Also choose a secret key k from the set of numbers $\{0, 1, \ldots 25\}$; for this problem, let's pick our secret key to be $k = 15$.

– Now, take the message you want to send: for example, let's send the phrase $\boxed{\text{chain fusion}}$.

– Swap the letters for numbers, to get $\boxed{2.7.0.8.13 \quad 5.20.18.8.14.13}$.

– Take each number in your code and add our secret code to it: this gives us $\boxed{17.22.15.23.28 \quad 20.35.33.23.29.28}$.

– Now replace each number in our code with its value modulo 26; this gives us $\boxed{17.22.15.23.2 \quad 20.9.7.23.3.2}$.

– Translate this back to letters, to get the encrypted message $\boxed{\text{rwpxc ujhxdc}}$. To an outsider, this would look like gibberish! In ancient Roman times, most people who would intercept such a message would assume that it was written in a foreign language and not be able to translate it.

– But, if you knew the secret key, you could just translate this message back to numbers and subtract 15 from each letter modulo 26. This would give you the original message, as desired.

This specific cipher is easy to crack: a simple brute-force approach of trying all 26 keys one-by-one on the encrypted text would lead you to the right answer pretty quickly. (Try it out by hand or by writing a program; it's not too bad.)

There are better ways to keep information secure, though! See courses like Compsci 110 and Maths 328 for some more modern approaches (that still use modular arithmetic at their core!)

1.5 % and Arithmetic

Consider the following problem:

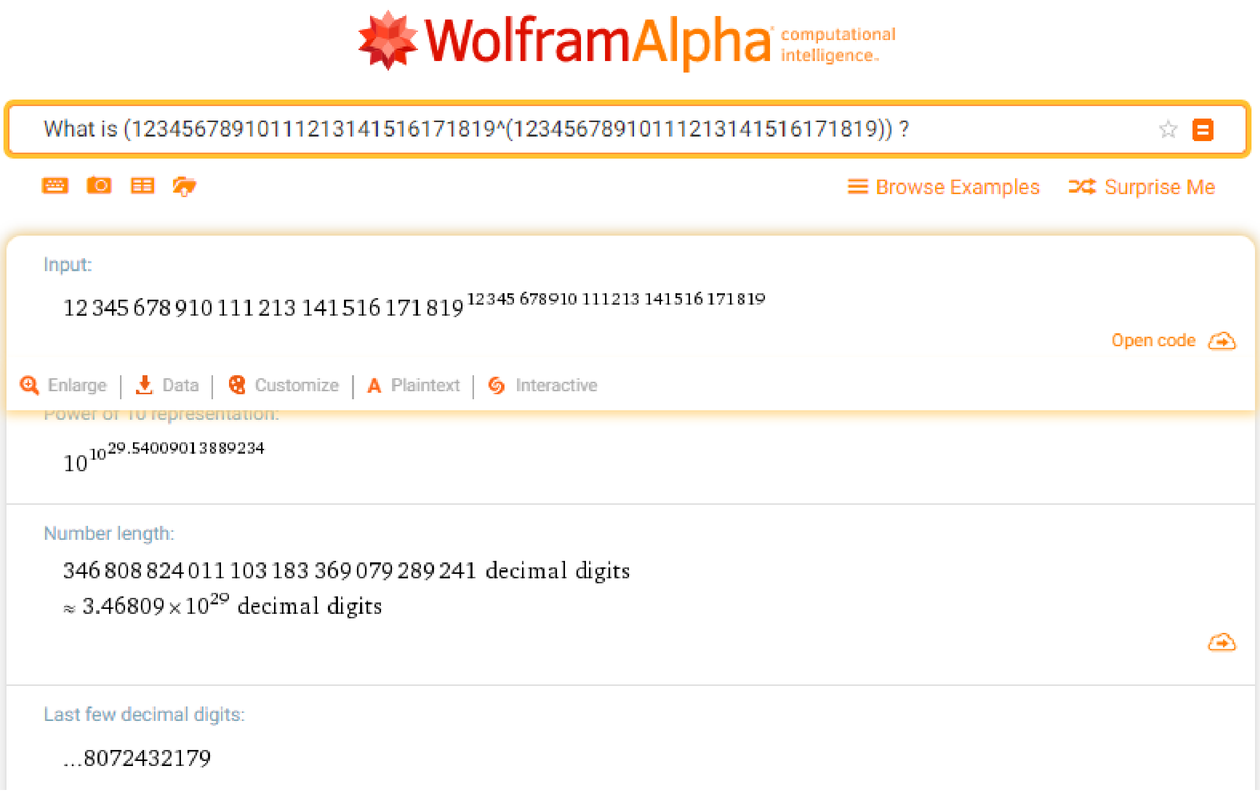
Question 1. *What is*

$$12345678910111213141516171819^{12345678910111213141516171819}\,?$$

If you tried to use a computer to directly expand this problem, the size of the resulting number (approximately $10^{10^{29}}$) would take up $\approx 10^{17}$ terabytes, i.e. several orders of magnitude in excess of the current total storage capacity of humanity.

As of 2019, according to a variety of articles + some back-of-the-envelope estimation by the coursebook's authors. By way of comparison, the total data storage of humanity was about 10^8 terabytes in 2008. To put that figure in modern perspective: YouTube broadcasted more than this much data in the past six months.

And yet, if you pop over to WolframAlpha and ask it this question, you'll get the following answer in about three seconds:


What is (12345678910111213141516171819^(12345678910111213141516171819)) ?

Input:

$12\,345\,678\,910\,111\,213\,141\,516\,171\,819^{12\,345\,678\,910\,111\,213\,141\,516\,171\,819}$

Power of 10 representation:

$10^{10^{29.54009013889234}}$

Number length:

346 808 824 011 103 183 369 079 289 241 decimal digits

$\approx 3.46809 \times 10^{29}$ decimal digits

Last few decimal digits:

...8072432179

So: how is WolframAlpha able to calculate this so quickly, if the precise result is so staggeringly huge?

Partly, this is because the "powers-of-ten" approximation can be made by replacing the values here with just large powers of 10: i.e. things like $12345678910111213141516171819 \approx 10^{28}$ can make approximating things a bit easier. However, WolframAlpha **also** seems to know the last few digits of this number, which seems a little bit spooky: how can you calculate this without understanding the entire number?

The trick here is the **modulus operation**, which we can use to perform certain arithmetic tasks very quickly. To show how this is possible, we introduce the closely related idea of **congruency** modulo n:

Definition 1.8. *Take any three integers a, b, n. We say that a* ***is congruent to*** *b* ***modulo*** *n, and write $a \equiv b \bmod n$, if $a - b$ is a multiple of n.*

The phrase "a is equivalent to b modulo n" is also frequently used for this concept.

To get an idea for how this works, we calculate a number of examples here:

Example 1.9.

- $21 \equiv 5 \bmod 8$; this is because $21 - 5 = 16 = 2 \cdot 8$ is a multiple of 8.
- $21 \equiv 13 \bmod 8$; this is because $21 - 13 = 8$ is a multiple of 8. Notice that it's possible for a number to be congruent to many other possible values: i.e. 21 was congruent to both 5 and 13 modulo 8!
- $-19 \equiv 7 \bmod 2$; this is because $-19 - 7 = -26 = (-13) \cdot 2$ is a multiple of 2.
- $14 \not\equiv 18 \bmod 5$; this is because $14 - 18 = -4$ is not a multiple of 5.
- For any two integers a, b, $a \equiv b \bmod 1$; this is because $a - b$ is always a multiple of 1 (as any integer is a multiple of 1!)

- For any two integers a, b, we have $a \equiv b \bmod 2$ if and only if a, b are both even or a, b are both odd. This is because $a \equiv b \bmod 2$ holds if and only if $a - b$ is a multiple of 2, i.e. $a - b$ is even, and this only happens when a, b are the same **parity**: i.e. when they're both even or both odd.

Congruency and our modulus operator are very closely linked:

In English/mathematics, we say that two statements are **equivalent** if they hold in precisely the same situations: i.e. whenever one is true, the other is true, and vice-versa. For example, the two statements "n is an even prime number" and "$n = 2$" are equivalent, as they both describe the same thing!

Claim 1.8. *Take any three values a, b, n such that $n \neq 0$. Then the following two statements are equivalent:*

1. $a \,\%\, n = b \,\%\, n$.
2. *$a - b$ is a multiple of n; i.e. $a \equiv b \bmod n$.*

We reserve proving this claim for your practice problems at the end of this chapter (see exercise 7). Instead, let's talk about how we can use this:

Claim 1.9. *Suppose that a, b, c, d, n are any set of integers with $n \neq 0$, such that $a \,\%\, n = b \,\%\, n$ and $c \,\%\, n = d \,\%\, n$.*

Then we have the following properties:

- $(a + c) \,\%\, n = (b + d) \,\%\, n$.
- $(ac) \,\%\, n = (bd) \,\%\, n$.

This claim is basically just saying that we can do "arithmetic modulo n!" That is: for numbers a, b, c, d, you know that if $a = b, c = d$ then $ac = bd$ and $a + c = b + d$, by just combining these equalities with the addition and multiplication operations. This claim is saying that if your values are "equal modulo n," the same tricks work!

We prove this claim here:

Proof. We need to prove our claim in general here, as we're making a claim about any possible set of values, not a specific set of values! To do so, we let a, b, c, d, n be any set of integers such that $a \,\%\, n = b \,\%\, n$ and $c \,\%\, n = d \,\%\, n$.

If we now use Claim 1.8, we get the following: $a - b$ is a multiple of n, and $c - d$ is a multiple of n. By definition, this means that we can write $a - b = kn$ and $c - d = ln$ for some pair of integers k, l.

By adding these equations together, we get $(a + c) - (b + d) = kn + ln = (k + l)n$; that is, that $(a + c) - (b + d)$ is a multiple of n. This means that, by Definition 1.8, $a + c \equiv b + d \bmod n$! Applying Claim 1.8 gives us $(a + c) \,\%\, n = (b + d) \,\%\, n$, as claimed.

If we take $a - b = kn$ and multiply both sides by c, we get $ac - bc = kcn$; similarly, if we take $c - d = ln$ and multiply by b we get $bc - bd = bln$. Adding these equations together gives us $ac - bc + bc - bd = kcn + bln = (kc + bl)n$; i.e. $ac - bd$ is a multiple of n. This means that $ac \equiv bd \bmod n$! Again, applying Claim 1.8 then tells us that $(ac) \,\%\, n = (bd) \,\%\, n$, as desired. □

More English/mathematics word definitions! A **corollary** is a claim that follows quickly from the claim that just came before. That is, a corollary is a "consequence" of the earlier claim, or something that you get "for free" once you've proven some earlier result.

A quick corollary to Claim 1.9 is the following:

Corollary 1.1. *If $a \,\%\, n = b \,\%\, n$, then for any positive integer k, we have $(a^k) \,\%\, n = (b^k) \,\%\, n$.*

We leave proving this for Exercise 9; try proving this by using the multiplication property in Claim 1.9!

Instead of spending more time with this sort of abstract stuff, let's do something **practical**: let's talk about how we could solve our WolframAlpha problem! Specifically, let's solve the following problem:

Claim 1.10. *The last digit of* 213047^{129314} *is* 9.

Proof. The clever thing here is not that we can calculate this (again, if we just wanted an answer, we could plug this into WolframAlpha), but rather that we can calculate this *easily*! That is: we can use modular arithmetic to find this number without needing a calculator, with relatively little work overall.

To do so, just make the following observations:

- First, $213047 \% 10 = 7$, and in general any positive number is congruent to its last digit modulo 10. This is by definition: if you take any number a greater than 10, subtracting 10 from it does not change its last digit! Therefore, the process we defined to calculate $a \% 10$ will never change the last digit of a, and thus its output at the end is precisely a's last digit.
- Therefore, we have that $\left(213047^{129314}\right) \% 10 = \left(7^{129314}\right) \% 10$, by using our "exponentiation" result from earlier.
- Now, notice that $\left(7^2\right) \% 10 = 49 \% 10 = 9$, and thus that $\left(7^4\right) \% 10 = \left(7^2 \cdot 7^2\right) \% 10 = (9 \cdot 9) \% 10 = 81 \% 10 = 1$.

  As a result, we have that for any k, $\left(7^{4k}\right) \% 10 = (7^4)^k \% 10 \equiv 1^k \% 10 = 1$.
- Therefore, because $129314 = 129300+12+2$, and any multiple of 100 is a multiple of 4, we have that $\left(7^{129314}\right) \% 10 = \left(7^2 \cdot 7^{\text{a multiple of 4}}\right) \% 10 = 9 \cdot 1 = 9$.

  In other words, our number's last digit is $\boxed{9}$!

□

It's worth noting that this trick isn't just useful for humans, but for computers as well: you can use tricks like this to massively speed up calculations in which you only care about the number's last few digits, and/or other pieces of partial information.

We can use this trick to basically calculate *any* number to *any* other number's last digit! For example, the task of finding the **last digit** of

$$12345678910111213141516171819^{12345678910111213141516171819}$$

is now something we can do very quickly:

- Like before, we only care about the last digit of the thing we're exponentiating: i.e. we can reduce this problem to just finding $9^{123\ldots19}$'s last digit.
- $\left(9^2\right) \% 10 = 81 \% 10 = 1$; therefore, for any even number $2k$, we have $\left(9^{2k}\right) \% 10 = \left((9^2)^k\right) \% 10 = 1^k = 1$.
- Therefore, for any odd number $2k+1$, we have $\left(9^{2k+1}\right) \% 10 = \left(9 \cdot 9^{2k}\right) \% 10 = 9 \cdot 1 = 9$.
- So, because the last digit of our base is 9 and the exponent is odd, the last digit of our entire number is $\boxed{9}$!

Generalizations of this process (i.e. working modulo 100 to find the last two digits, etc) can be used to quickly find the last few digits of any number. This can be quite useful when writing computer algorithms: most of the time when working with large numbers, you only need the first few and last digits for an approximation of its size, not the entire exact value!

1.6 Other Number Systems

In the past five sections, we've focused on studying the integers; i.e. whole numbers. However, there are lots of problems in real life whose answers cannot be given with just integers alone. Even going back to primary school, you've seen and solved problems like the following:

1. Suppose that three hedgehogs come across two sausages while wandering through a backyard barbie, and want to share them equally. How many sausages should each hedgehog get?
2. If you have one and a half pavlovas left over from the barbeque, and you eat a third of one for breakfast, how much pav do you have left?
3. How many dollars are there in ten cents?

Note: pavlovas are likely not part of a balanced breakfast.

The answers to each of these $\left(\frac{2}{3}, \frac{3}{2} - \frac{1}{3} = \frac{7}{6} \text{ and } \frac{10}{100} = \frac{1}{10}, \text{ respectively}\right)$ are all relatively easy to calculate, and are tasks you've seen in school for years now! However, their solutions are not things that we can express as integers.

To work with them, we need some new sets of numbers! We start with one such set that you're already familiar with:

Definition 1.9. *A **rational number** is any number that we can express as a ratio $\dfrac{x}{y}$, where x, y are integers and y is nonzero. We let $\mathbb{Q}$ denote the collection of all rational numbers.*

Definition 1.10. *Given a rational number x, we say that x is the **numerator** of $\frac{x}{y}$, and that y is the **denominator**.*

Example 1.10. The numbers $\frac{1}{2}, \frac{-2}{13}, \frac{3}{6}$ and $\frac{0}{7}$ are all rational numbers. The numerator of $\frac{3}{6}$ is 3, and the denominator is 6; similarly, the numerator of $\frac{0}{7}$ is 0, and the denominator is 7.

Notice that every integer x can be written as a rational number, because $x = \frac{x}{1}$.

There are several operations we know how to perform on rational numbers:

Fact 1.1. *If $\frac{a}{b}, \frac{c}{d}$ are a pair of rational numbers, then we can add and multiply them! Specifically, we say that*

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}, \qquad \textit{and} \qquad \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}.$$

Notice that these are both still rational numbers, as their numerator and denominators are still integers, and the denominator is nonzero because $b, d \neq 0$.

With these operations in mind, we can answer our second exercise:

Answer to Exercise 1.2. In this problem, we're asking if it is ever possible for $\frac{1}{x} + \frac{1}{y}$ to be equal to $\frac{1}{x+y}$.

If you're ever given a problem like this (i.e. someone hands you an equation and asks you "are there any solutions,") the first thing you should try to do is **solve the equation**! That is: try to rearrange the equation for one of your variables in terms of the other and/or some constants, and hope that this tells you something.

In this situation, we know that x, y and $x + y$ must be nonzero if the fractions $\frac{1}{x}, \frac{1}{y}, \frac{1}{x+y}$ are defined. Therefore, the numbers $\frac{1}{x}, \frac{1}{y}$ are rationals, and so their sum is just $\frac{x+y}{xy}$ as noted above.

If $\frac{x+y}{xy} = \frac{1}{x+y}$, then because x, y and $x+y$ are all nonzero, we can multiply both sides by the denominators (i.e. multiply both sides by $xy(x+y)$.) Doing so will get rid of our fractions, as

$$\begin{aligned}(xy)(x+y)\left(\frac{x+y}{xy}\right) = (xy)(x+y)\left(\frac{1}{x+y}\right) \quad &\Rightarrow \quad (x+y)^2 = xy \\ &\Rightarrow \quad x^2 + 2xy + y^2 = xy \\ &\Rightarrow \quad x^2 + xy + y^2 = 0.\end{aligned}$$

This is a simpler equation! In particular, if we think of x as our variable and y as a constant, we have a **quadratic equation**. In general, a quadratic equation of the form $ax^2 + bx + c = 0$ has the two solutions $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ when b^2-4ac is nonnegative, and has no real-valued solutions if $b^2 - 4ac$ is negative.

In this situation, the quadratic formula above tells us that $a = 1, b = y, c = y^2$, and thus that our solutions (if they exist) would be $\frac{-y\pm\sqrt{y^2-4y^2}}{2}$. However, we know that $y^2 - 4y^2 = -3y^2$ is always a negative number; this is because -3 is negative, while y^2 is a nonzero number squared (and thus is positive.) Therefore we have no solutions! ☐

There are other number systems that you've likely seen before in high school as well:

Definition 1.11. *The **natural numbers**, denoted $\mathbb{N}$, is the collection of all nonnegative integers. That is, $\mathbb{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$.*

Definition 1.12. *The **real numbers**, denoted $\mathbb{R}$, is the collection of all numbers that you can write out with a (possibly infinite) decimal expansion: i.e. it's the collection of things like*

- 2.1,
- -724,
- $0.111111\ldots = 0.\overline{1}$, *and*
- $-3.1415926535\ldots$

So, for example, the real number contain all of the rational numbers, because you can do things like write

- $\frac{1}{2} = 0.5$,
- $\frac{1}{3} = 0.333333 = 0.\overline{3}\ldots$, and
- $\frac{22}{7} = 3.142857142857142857\ldots = 3.\overline{142857}$.

However, there are also real numbers that are **not** rationals: i.e. there are quantities out there in the world that we cannot express as a ratio of integers! We call such numbers **irrational**.

Observation 1.4. *Notice that every real number, by definition, is either rational or irrational.*

It's a bit beyond the skills we have at the moment, but numbers like

$$\pi = 3.14159265358979323846264338327950288419716939\ldots$$

and

$$e = 2.71828182845904523536028747135266249775724709\ldots$$

are both irrational numbers. Later on in this class, we'll prove that $\sqrt{2}$ is also an irrational number (check out the proof by contradiction section of this book if you cannot wait!)

With that said, though, it would be a bit of a shame to not show you at least one irrational number. So, let's close this chapter by doing this!

Claim 1.11. *The number* $\log_2(3)$ *is not rational.*

Proof. We prove this claim by using the "suppose we're wrong" structure we've successfully used in Claims 1.6 and 1.3. That is: what would happen if $\log_2(3)$ *was* rational?

Well: we could write $\log_2(3) = \frac{x}{y}$, for two integers x, y. Note that because $\log_2(3)$ is positive (it's greater than $\log_2(2) = 1$), we can have $x, y > 0$.

Exponentiating both sides of this equation by 2 gives us $2^{\log_2(3)} = 2^{x/y}$. By definition, $\log_2(\star)$ and $2^\star$ cancel each other out, so this simplifies to $3 = 2^{x/y}$.

Now, raising both sides to the y-th power gives us $3^y = \left(2^{x/y}\right)^y$. By using our known rules for exponentiation, this simplifies to $3^y = 2^x$.

On the left-hand-side, we have an odd number: it's just y copies of 3 multiplied together! On the right-hand-side, however, we have an even number: it's x copies of 2 multiplied together.

These two values cannot be equal! In other words, our initial assumption that we could write $\log_2(3) = \frac{x}{y}$ must have been flawed, as this assumption would imply that a number can be even and odd at the same time. Therefore $\log_2(3)$ is irrational, as desired. □

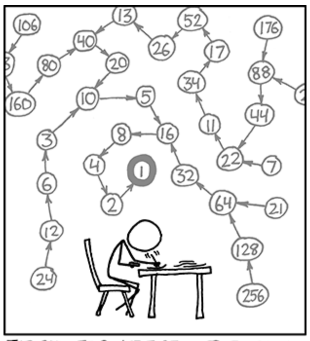
We close this chapter with some exercises for you to try out. Give these an attempt, and post your answers / attempts on Piazza!

1.7 Practice Problems

1. (-) Let m be even. Show that $-m$ is also even.
2. Show that if a and b are both odd, then ab is also odd
3. (+) Show that if a is an even number and k is a positive integer, then a^k is even.
4. In Claim 1.3, we showed that no number is both even and odd at the same time. Prove the converse of this claim: that is, prove that every integer is either even or odd (i.e. you can't have an integer that is both not even and not odd.)
5. (-) Write down all of the numbers between 1 and 100 that are congruent to 2 modulo 12.
6. How many numbers between 1 and 1000 are congruent to 0 modulo 3? How many are congruent to 0 modulo 4? Can you make a formula to count how many numbers between 1 and 1000 are congruent to 0 modulo n, that works for any positive integer n?
7. (+) Prove Claim 6.2.
8. Take any three-digit number abc (where c is the ones digit, b is the tens digit, and a is the hundreds digit.) Show that $abcabc$ is never a prime number.
9. Prove Corollary 1.1.
10. (+) Suppose that $p_1, p_2, \ldots p_n$ are all prime numbers. Must the number $N = 1 + (p_1 \cdot p_2 \cdot \ldots \cdot p_n)$ be prime? Either explain why, or find a set of prime numbers such that the value N defined above is not prime.
11. (-) Show that $n^2 - n$ is always even, for any integer n.
12. Show that if n is an odd number, then $n^2 - 1$ is a multiple of 8.
13. Find the last digit of 12345^{67890} without using a computer or calculator.

14. (+) Find the last digit of 9876^{54321} without using a computer or calculator.
15. (+) What is the longest English word that can be shift-ciphered into another English word?
16. (++) Take an integer n. If it's odd, replace it with $3n+1$; if it's even, replace it with $n/2$. Repeat this until the number is equal to 1. Will this process always eventually stop?
17. Consider the following claim: "Let $\frac{a}{b}, \frac{c}{d}$ be a pair of rational numbers. Suppose that $ad > bc$. Then we have $\frac{a}{b} > \frac{c}{d}$."

    Is this claim true? If it is, explain why. If it is not, find a pair of fractions that demonstrate that this claim is false.
18. If x is rational and y is irrational, must $x+y$ be irrational? Either explain why or find a counterexample (i.e. find a pair of numbers x, y such that x is rational, y is irrational, and $x+y$ is rational.)
19. If x and y are both irrational numbers, must $x+y$ be irrational? Either explain why or find a counterexample (i.e. find a pair of numbers x, y such that x, y are irrational, but $x+y$ is rational.)
20. (+) Show that if a, b, c are all odd integers, then there is no rational number x such that the equation $ax^2 + bx + c = 0$ holds.
21. (++) Is $\pi + e$ irrational?



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

(from https://xkcd.com/710/)

Compsci 120 University of Auckland

Sets and Strings

2020 *Chapter 2*

Exercise 2.1. *You're trying to break into a safe that has a PIN lock. The safe has two buttons: 0 and 1. The PIN you're trying to guess is a three-digit sequence of binary numbers, and accepts the last three digits you've typed in without needing you to hit enter: i.e. if you typed in "00010," the safe would open if the pin was either "000" or "001" or "010".*

Sounds easy, right? There's only eight possible PINs to check (two possibilities per digit, three digits in the PIN $\Rightarrow 2^3 = 8$ *possible pins), so we should be able to brute-force the lock by checking all possibilities.*

However, the safe is wired to call the cops if more than ten buttons are pressed and the correct PIN is not entered. As such, we can't use our brute-force approach: that could take $8 \cdot 3$ *entries!*

Is there an approach that is guaranteed to break us into the safe?

Exercise 2.2. *You're a geneticist! As such, you're working with DNA strands, which we can think of as long strings over the alphabet* $\{A, C, G, T\}$*, if we let these letters represent the nucleotides adenine, cytosine, guanine and thymine.*

You've designed a clever little combination of DNA restriction+polymerase enzymes that do the following: given any string s of DNA strands, every time there's a substring of the form "... AC ... " in s, that substring gets cut out and replaced with "... CCA ... "

So, for example, if your DNA strand was "ACGT," it would get turned into "CCAGT" and then would stay stable from there. If your strand was "ACCT", however, it would first turn into "CCACT", and then "CCCCAT."

Suppose you're originally working with strings of DNA all of the form "AAAAC," and you dump them into a bath with your enzymes in it. What would you expect to see at the end of this process?

Earlier in this coursebook, we discussed various properties about numbers (divisibility, modular arithmetic, etc) that are very useful in computer science!

However, numbers are not the only things that we work with in computing systems. We also work heavily with things like passwords, user IDs, databases full of names: i.e. **strings**! We study these objects in the following section:

2.1 Strings

First, let's define what an **alphabet** is:

Definition 2.1. *An* ***alphabet*** *is any collection of symbols.*

Example 2.1. You're already familiar with the Roman alphabet:

$$\boxed{\text{A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z}}$$

Another alphabet that comes in handy is the collection of decimal digits! We use this to describe numbers:

$\boxed{0,1,2,3,4,5,6,7,8,9}$

If we're working in binary, we use a much smaller alphabet:

$\boxed{0,1}$

If we're working with DNA, we'd use

$\boxed{A,C,G,T}$

as our alphabet, where these represent the four nucleotide bases cytosine [C], guanine [G], adenine [A] or thymine [T].

There are other alphabets that are too big to write down here: for example, the set of all Unicode symbols, or the set of all emojis!

Given an alphabet, it's often useful to be able to refer to the whole thing with a symbol. We'll do this by writing something like $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$. This notation, where we list our symbols between a pair of curly braces and separate them with commas, tells us that Σ is an alphabet containing the ten symbols 0,1,... 9.

With the definition of an alphabet in hand, we can define strings:

Definition 2.2. *Take any alphabet Σ. A **string** over the alphabet Σ is any sequence of letters in an alphabet.*

Some people refer to strings as "words:" if you see an author referring to a collection of words over a given alphabet, this is just a synonym for strings!

Example 2.2. If we let Σ be the Roman alphabet described earlier, then "cat," "mongoose," and "ssssssssssss" are all strings over this alphabet. Note that these strings don't have to correspond to any particular meaning; they're just sequences of symbols!

If we let Σ be the decimal alphabet, then "123," "00012," and "999" are each possible strings over this alphabet. Again, these don't always have to correspond to numbers! In particular, notice that as **strings** we think that "00012" and "12" are different things. Even though as numbers they're different, as strings they're quite different: "00012" has zeroes in it, while "12"does not. (That is, think about entering a password on your phone. There, if someone has a password of "00012," entering "12" shouldn't unlock your phone!)

We will sometimes not specify an alphabet, and instead just refer to strings by listing their entries. If so, we assume that their alphabet is the most reasonable one to work with that string in (usually either the Roman alphabet, decimal, or binary.)

A particularly useful string to refer to is the **empty string** "", i.e. the string containing no symbols. We denote this string by writing λ.

Strings are incredibly useful in computer science! Essentially every program we have works with data in the form of strings, in the form of ID numbers, names, IP addresses, and just simply the binary strings that encode literally everything that a computer does.

Perhaps the simplest operation to define on strings is **length**:

Definition 2.3. *The **length** of any string is the number of characters in that string.*

Example 2.3. The string "abcdef" has length 6, the string "00000" has length 5, and the string "0123" has length 4.

Note that when we're working with strings, writing something like "$s_1 s_2 s_3 \ldots s_n$" does *not* mean that we're multiplying these things all together as if they were numbers! That is: the string "0123" is not the same thing as the product $0 \cdot 1 \cdot 2 \cdot 3 = 0$. This is why it's important to keep track of the type of thing you're working with / in general, at the start of problems, to define your variables and notation.

The idea of length is useful when we're trying to describe a general string! Many arguments involving strings will start with the sentence "Take a string s over the alphabet Σ. Let n be the length of s, and write s as $s_1 s_2 s_3 \ldots s_n$."

In particular, we can use this to define what it means for two strings to be equal:

Definition 2.4. *Take any two strings $s = s_1s_2\dots s_n$ and $t = t_1t_2\dots t_n$ of the same length We say that s and t are* ***equal*** *if $s_1 = t_1, s_2 = t_2$, ... and $s_n = t_n$.*

In other words, two strings are equal if and only if they are literally character-for-character identical! Note that two strings of different lengths are always nonequal.

Example 2.4.

- The strings "00001" and "1" are different. Even though the underlying numbers they represent are the same, these are different-length strings.
- If we take the alphabet given by all characters on a keyboard, the strings "12+23" and "10+25" are different. Even though these are the same length and represent the same underlying integer, the characters are different in some places: for instance, the second character of the first string is 2, while the second character of the second string is 0.

A particularly useful operation on strings in computer science is **concatenation**:

Definition 2.5. *Take any two strings $s = s_1s_2\dots s_n$ and $t = t_1t_2\dots t_m$. The* ***concatenation*** *of s and t, written st, is the string $s_1s_2\dots s_nt_1t_2\dots t_m$.*

Example 2.5.

- Let $s =$"song" and $t =$"bird". Then st is the string "songbird".
- Let $s =$"12" and $t =$"0". Then st is equal to "120". Notice that this is very different to what we would mean by writing st if we thought of s, t as integers; there, st would denote $12 \cdot 0 = 0$!

  In general, if you're using string concatenation on strings of numbers, make sure to indicate this to your reader repeatedly through your working so that they know what you're doing. The use of quotation marks can help keep things clear: that is, because we wrote $s =$"12" and $t =$"0", we've told you that we are thinking of s, t as strings, and thus that concatenation is the appropriate way to combine them.
- You can concatenate multiple strings at once: i.e. if $s =$"3", $t =$".", and $u =$"14159265...", then stu is just "3.14159265..."

Notice that if s has length n and t has length m, then st has length $n + m$.

Concatenation is used in tons of practical applications:

- Every bank account number is a concatenation of a bank code (telling you what company you bank with,) an account number (which tells the bank who owns this account,) and an account type code (telling you what kind of account that number is attached to.)
- We saw that many ID numbers have "check digits" when we worked with modular arithmetic! As such, your full ID number is usually created by concatenating your account number with the check digit.

Related to the idea of **concatenation**, we have the concepts of **prefixes**, **substring** and **suffixes**:

Definition 2.6. *Let s and t be strings. We say that s is a* ***prefix*** *of t if t is just s with some additional stuff possibly tacked on the end: i.e. if we can find a third string u such that $su = t$.*

Similarly, we say that s is a ***suffix*** *of t if t is just s with some additional stuff possibly tacked on the front: i.e. if we can find a third string u such that $us = t$.*

Finally, we say that s is a ***substring*** *(alternately, an "infix") of t if t is just s with some stuff possibly tacked on both the front and end: i.e. if we can find strings u, v such that $usv = t$.*

Example 2.6.

- If t = "snowball," then "snow" is a prefix of t, "ball" is a suffix of t, and "now" is a substring of t.
- If t = "112323411," then "112" is a prefix of t, "323411" is a suffix of t, and "2" is a substring of t.

As a bit of practice with writing arguments, we study a few claims:

Claim 2.1. *The empty string λ is a prefix, suffix, and substring of every string t.*

Proof. Take any string t. Notice that $t = \lambda t = t\lambda = \lambda t \lambda$, because attaching the empty string to the start or end of any string doesn't change it. Therefore λ meets the definition of being a prefix, suffix, and substring for any other string t! $\square$

Claim 2.2. *If s is a prefix of t, then s is a substring of t.*

Proof. If s is a prefix of t, then there is some string u such that $su = t$. Therefore, we have $\lambda su = t$ as well, because concatenating the empty string λ with any string doesn't change it! This shows us that s satisfies the definition of substring, as claimed. $\square$

We can use this idea of a "substring" to answer our safe-cracking problem:

Answer to Exercise 2.1. Think of the sequence of keys we're entering into the safe as a string s. If we do this, then the properties we want s to have are the following: we want every three-digit binary string to occur as a substring of s, and we want s to have length at most 10.

As it turns out, we can do this! Enter the following string: "0001011100." This string has length 10, and contains all possible three-digit pins as subsequences, as shown in the margins. Success!

0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0

We can also use it to answer our DNA puzzle:

Answer to Exercise 2.2. On one hand, we could just brute-force the answer, by repeatedly looking for "AC" substrings and replacing them with "CCA" substrings:

AAA AC	→	AAA CCA,
AA AC CA	→	AA CCA CA,
A AC C AC A	→	A CCA C CCA A,
AC C AC CCAA	→	CCA C CCA CCAA,
CC AC CC AC CAA	→	CC CCA CC CCA CAA,
CCCC AC CCC AC AA	→	CCCC CCA CCC CCA AA,
CCCCCC AC CCCCAAA	→	CCCCCC CCA CCCCAAA,
CCCCCCCC AC CCCAAA	→	CCCCCCCC CCA CCCAAA,
CCCCCCCCCC AC CCAAA	→	CCCCCCCCCC CCA CCAAA,
CCCCCCCCCCCC AC CAAA	→	CCCCCCCCCCCC CCA CAAA,
CCCCCCCCCCCCCC AC AAA	→	CCCCCCCCCCCCCC CCA AAA,

In other words: the result is a string of 16 C's, followed by 4 A's.

Alternately, we could just notice the following: every time a C moves past an A, we replace that C with two C. Therefore, if we move a C past two A's in a row, we'd expect to repeat this "doubling" process twice, and have four C's; in general, if we move a C past n A's in a row, we'd expect to see 2^n C's at the end, as we've doubled our C's n times in this process! This matches our results, as $2^4 = 16$.

2.2 Sets

A second useful object, that we will often study in relation to strings, is the concept of a **set**:

Definition 2.7. *A* ***set*** *A is just a collection of things. We call those things the* ***elements*** *of A, and write $x \in A$ to denote with symbols the statement "x is an element of A."*

To describe a set, we just list its elements between a pair of curly braces: for example, $\{1, 2, 3\}$ would be how we would describe the set consisting of the three numbers 1, 2 and 3.

Basically every collection of things in real life can be thought of as a set:

Example 2.7.

- The collection of all strings in the Oxford English Dictionary is a set. It contains elements like "heart" and "number," but not things like "arbleorble."
- The collection of all words in Māori is a set. This set contains elements like "tapawhā" and "tau" (the Māori words for rectangle and number,) but does not contain strings like "123abc."
- The collection of all commands in C is a set.
- The collection of all binary strings of length at most 2 is a set. We could write this set out by listing its elements: $\{\lambda, 0, 1, 00, 01, 10, 11\}$.
- The "empty" set containing no elements $\{\}$ is a set! We call this the **empty set**, and refer to this by drawing the symbol $\varnothing$. This is a fairly useful set to be able to refer to, for the same reasons that 0 is a useful number; it can be handy to talk about "nothing" in a concrete way!
- The set of all prime numbers is a set: $\{2, 3, 5, 7, 11, 13, \ldots\}$
- The set of integers $\mathbb{Z}$, the set of rational numbers $\mathbb{Q}$, the set of natural numbers $\mathbb{N}$, and the set of real numbers $\mathbb{R}$ are all sets.
- The set of all polynomials with degree at most 3 is a set: it contains things like $2x - 4$ and $x^3 - 3x^2 + \pi$.
- The set of all irrational numbers is a set.
- The set of all numbers that are solutions to the equation $x^3 - 3x^2 + 3x - 1 = 0$ is a set. (Specifically, because $x^3 - 3x^2 + 3x - 1 = (x - 1)^3$, this set is just $\{1\}$, the set containing only one object, namely 1.)

Notice that sets can be finite (in the case of things like "the collection of all English words") or infinite (in the case of the set of all prime numbers!)

To make our lives easier when working with sets, let's make a few notational conventions about how we should treat them:

- When we're describing a set, we **don't care about the order** in which we list our elements: i.e. {cat, tag, tact} and {tag, cat, tact} are both the "same" to us! This is because we only care about what things are contained within a set; the order is something that we'll wind up changing a lot depending on the context (i.e. sometimes alphabetical, sometimes by length...) and isn't itself something we want to care about.

- Similarly, when we're describing a set, we only want to **list each element once**. This is because otherwise it would be quite irritating to try to look things up in our set: imagine a dictionary that just listed the word "mongoose" forty times in a row!

  As such, if someone gives you a set in which an element is repeated twice, we just remove duplicates: i.e. we say { cat, tag, tact, tact, tact } and {cat, tag, tact} are the same, and would never write the first thing if we couldn't help it.

- In the case of {cat, tag, tact}, we were able to describe our set by just listing its elements. This works for small cases, but becomes quite unwieldly for larger sets: imagine having to write out all of the words in French before discussing the French language!

  To deal with this, we have an alternate way of writing sets: you can describe them **by giving a property**. For instance, when we say "the set of all words in Māori" above, we're giving you a property that a given string of letters may or may not satisfy (i.e. "is it a word in Māori"), and then taking the set of all words that satisfy that property.

  While the sentences we used in our examples above do work as definitions for sets, you can also use the following more "math-y" construction: to describe the set of all strings s with property *blah*, you can just write

  $$\{s \mid s \text{ has property } blah\}.$$

  For instance, the set of all odd-length binary strings could be described as the following:

  We use the notation "∈" as shorthand for the word "in."

  $$\{s \mid \text{length}(s) = 2k + 1 \text{ for some } k \in \mathbb{Z}, \text{ and } s \text{ is a binary string.}\}$$

  The "s" on the left tells you the variable name, the divider | just separates the variable from its property, and the text at the right gives the required property.

  You can also use the left-hand part to describe the structure of your set's elements: i.e. something like

  $$\{\text{concatenate(“001”, s)} \mid \text{s is a binary string}\};$$

  gives you all binary strings that start with the prefix "001."

One useful concept when working with sets is a notion of "size:"

In maths, the word **cardinality** is used to refer to the size of a set. If you take papers like Maths 190 or Compsci 225, you can learn to study the idea of "different sizes of infinity" by working with cardinality! In particular, using the idea of a *bijection* in those courses, you can show that the integers, rationals, and natural numbers somehow all have the *same* "countable" size of infinity, while the real numbers somehow have a larger and "uncountable" size of infinity. . .

Definition 2.8. *A set A has* ***size*** *n if it contains precisely n different elements. If A contains infinitely many different elements, we say that A has "infinite" size. We denote the size of A by writing $|A|$.*

Example 2.8.

- The set $\{1, 2, 3, \pi, 7\}$ has size 5.
- The set of all binary strings of length 2, i.e. $\{00, 01, 10, 11\}$, has size 4.

Another useful concept when working with sets is the idea of a "subset:"

Definition 2.9. *Take two sets A, B We say that B is a* ***subset*** *of A, and write $B \subseteq A$, if every object in B is also an object in A.*

Example 2.9.

- Let A be the collection of all University of Auckland ID numbers, and let B be the collection of all University of Auckland ID numbers corresponding to active Compsci 120 students. Then B is a subset of A!
- Let A be the set of all binary strings of length 3, and let B be the set of all binary strings with exactly two 1's.

  Then B is **not** a subset of A. This is because B contains things like "11000", which are not in A. Similarly, A is not a subset of B, because A contains things like "000" that are not in B!
- Let A be the English language, and B be the collection of all English words that rhyme with "avocado." Then B is a subset of A, as every word in B is by definition a word in A!

We also have a number of useful operations that we perform on sets:

Definition 2.10. *Let A, B be a pair of sets. We define the* ***union*** *of these two sets, $A \cup B$, to be the collection of all elements that are in either A or B or both.*

Example 2.10.

- Let A be the collection of all English words with even length and B be the collection of all English words with odd length. In this case, $A \cup B$ is the collection of all English words.
- Let A be the collection of all Compsci 120 students that turned in assignment 1, and B be the collection of all Compsci 120 students that attended tutorial 1. Then $A \cup B$ is the collection of all Compsci 120 students who either attended tutorial 1 or turned in assignment 1, or both.

  In general, unions work like "or" operations: the union of a set defined by property A with a set defined by property B is just the collection of all elements that satisfy property A **or** B.
- Let A be the collection of the 1000 most common phrases used in spam emails (things like "You be a Winner!!!1!!") and B be a collection of dodgy email addresses (e.g. "`bill.gates@micr0soft.ie`"). Then, the union $A \cup B$ is a good start for a "block list," i.e. something that an email filter can use to automatically trash certain emails.

Definition 2.11. *Let A, B be a pair of sets. We define the* ***intersection*** *of these two sets, $A \cap B$, to be the collection of all elements that are in both A and B at the same time.*

Example 2.11.

- Let A be the English language and B be the German language. Then $A \cap B$ is the set of words that are both in English and German at the same time: i.e. words like "alphabet," "computer" and "tag" would be in $A \cap B$, as they are all both English and German words.
- Let A be the set of numbers that are multiples of 3, and B be the set of numbers who are multiples of 2. Then $A \cap B$ is the set of numbers that are multiples of both 2 and 3; i.e. it's the set of all numbers that are multiples of 6!

  Like how union was an "or," intersection works like an "and" operation: that is, the intersection of a set defined by property A with a set defined by property B is just the collection of all elements that satisfy property A **and** B.

- If A is the set consisting of ID numbers of current Compsci 120 students, and B is the set consisting of ID numbers of current Compsci 720 students, then $A \cap B = \varnothing$, the empty set. (This is because there are no students simultaneously taking 120 and 720!)

Definition 2.12. *Let A, B be a pair of sets. We define the **difference** of these two sets, written $A \setminus B$ or alternately $A - B$, to be the collection of all elements that are both in A and **not** in B at the same time.*

Example 2.12.

- If A was the set of ID numbers for all current Compsci 120 students, and B was the set of ID numbers for Compsci 120 students who attended at least eight tutorials, then $A \setminus B$ is the set of ID numbers for students who attended seven or fewer tutorials (i.e. the ID numbers of students who will not have perfect marks for tutorials. Don't be in this set!)
- If A is the set of prime numbers, and B is set of odd integers, then $A \setminus B$ is the collection of all primes that are not odd: that is, $A \cap B = \{2\}$.
- Let A denote the set of all ASCII strings of length at least 10, B be the set of all English words, and L_3 be a list of the 10,000 most common passwords. The set $A \setminus (B \cup L_3)$ is a good start to a list of "acceptable" passwords: i.e. if you were making a login system, you could require all of your users to pick words in $A \setminus (B \cup L_3)$. Doing this would mean that they have to pick passwords that
  - Have length at least 10 (i.e. are in A),
  - Aren't in a dictionary (i.e. not in B), and
  - Aren't commonly used (i.e. not in L_3).

  Useful!

Finally, we describe what it means for two sets to be **equal**:

Definition 2.13. *We say that two sets A, B are **equal** if they both consist of the same elements; that is, if*

- *Every element in A is a element in B, and*
- *Every element in B is also a element in A.*

If you go back to our remarks earlier, this should make sense. We said that the only thing we cared about for a set was the elements it contained; i.e. we didn't care about the order, and we ignored repeats/etc. Therefore, two sets should be the same if they contain the same elements!

A useful proof technique, that we'll often use to show that two sets are the same is the following. Take two sets A, B that you want to show are equal. Suppose you showed that

1. every element in A is a element in B, and also
2. every element in B is a element in A.

Then, by the definition above, we would know that A and B are equal! As such, we can use this two-part approach to prove that many pairs of objects are equal. We study a few examples here, to get the hang of this:

Claim 2.3. *Let A, B be any two sets such that $A \subseteq B$. Then $A \cup B = B$.*

Proof. We proceed as suggested above:

1. First, we show that every element in $A \cup B$ is in B. To do this, we note that by definition $A \cup B$ is the set of all elements that are in either A or B. Therefore, if we take any element $s \in A \cup B$, we either have $s \in A$ or $s \in B$. This lets us work in cases:

   - If $s \in A$, then recall that we've assumed that $A \subseteq B$. By definition, this means that every element in A is also in B. Therefore, we have $s \in B$.
   - If $s \in B$, then we trivially have $s \in B$.

   Therefore, we've shown that for any $s \in A \cup B$, we have $s \in B$, as desired.

2. Second, we show that every element in B is in $A \cup B$. This is not too challenging.

   Just notice that $A \cup B$, by definition, contains all elements in either A or B. Therefore, for any $s \in B$, we have $s \in A \cup B$ by definition.

This completes the two-way argument, as desired! □

This is not the only way to prove that two sets are equal! As always, simply expanding the definitions of both sets can often do the same trick:

Claim 2.4. *Let A, B be any two sets. Then $(A \setminus B) \setminus A = \varnothing$.*

Proof. We proceed by expanding our definitions:

1. First, notice that $A \setminus B$, by definition, is the collection of all elements in A that are not in B.
2. By definition again, $(A \setminus B) \setminus A$ is "(the collection of all elements in A that are not in B) that are also not in A."
3. We can simplify this to "the collection of all elements in A that are not in B or A."
4. Every element of A is, um, in A.
5. Therefore, "the collection of all elements in A that are not in B or A" is the **empty set**, as the "not in A" condition eliminates all of the elements in A.

So we've proven our claim! □

To close our chapter, we study a trickier example of this process:

Claim 2.5. *If A, B, C are three sets, then $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$.*

Proof. We again proceed by expanding our definitions:

- On the left-hand-side, we have $A \setminus (B \cup C)$. By definition, $A \setminus (B \cup C)$ consists of all of the elements that are in A, but not in $B \cup C$.

  As well, by definition we know that $B \cup C$ is the set of all elements that are in either B or C, or both.

  Therefore, $A \setminus (B \cup C)$ can just be thought of as all of the elements that are in A, but not in either B or C.

- On the right-hand-side, we can similarly use our definitions to notice that $A \setminus B$ is the set of all elements that are in A but not B, and that $A \setminus C$ is the set of all elements that are in A but not C.

  As a consequence, we have that $(A \setminus B) \cap (A \setminus C)$ is the set of all elements that are both (in A but not B) **and** (in A but not C). Logically, we can simplify this sentence to the condition "in A, but not in either B or C."

These are the same statements; therefore we've shown that these sets are equal! □

2.3 Practice Problems

1. (-) Show that any suffix of a word t is a substring of t.
2. If s is both a prefix and a suffix of t, then must $s = t$? Either show that this is true, or find a counterexample.
3. Show that if s is a substring of t and t is a substring of s, then $s = t$.
4. (+) Suppose our safe in Exercise 2.1 had PIN numbers with decimal digits, not binary (i.e. they could be any digit from 0-9, instead of just 0 and 1.)

   What is the smallest number of buttons we would have to press to guarantee that the safe would open in this situation?
5. Suppose that A and B are two sets with the following property: every string in A is a prefix of a string in B. Is it possible for A to contain more elements than B? Either find such an example, or explain why this is impossible.
6. (-) Explain why $\varnothing \cup L = L$ for any set L.
7. Show that for any three set A, B, C, that $A \smallsetminus (B \cap C) = (A \smallsetminus B) \cup (A \smallsetminus C)$. (Try doing so using the method from Claim 2.3, and then try with the method from Claim 2.4! Which do you prefer?)
8. Take two binary strings s, t of the same length. We say that s and t are **orthogonal** if they disagree at precisely half of their locations: for example, $s =$"1111" and $t =$"1100" are orthogonal.
   (a) (-) Show that if s, t are odd-length strings, then s and t cannot be orthogonal.
   (b) Find a set consisting of four length-4 strings that are all orthogonal to each other (i.e. every possible pair of strings in your set should be orthogonal.
   (c) (+) Find a set consisting of 2^n length-2^n strings that are all orthogonal to each other.
   (d) (++) What is the largest set of orthogonal length-668 strings that you can make?

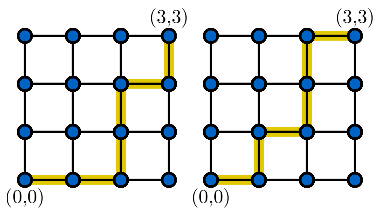
Compsci 120	University of Auckland
Combinatorics and Probability	
2020	*Chapter 3*

3.1 How to Count

This might seem like a silly section title; counting, after all, is something that you learned how to do at a very young age! So let's clarify what we mean by "counting."

On one hand, it is easy to see that there are four elements in a set like

$$A = \{3, 5, 7, \text{Snape}\}.$$



We're not going to practice counting things like this! Instead, consider the following four exercises:

Exercise 3.1. *How many strings of five letters are* ***palindromes*** *(i.e. can be read the same way forwards and backwards?)*

Exercise 3.2. *A* ***lattice path*** *in the plane $\mathbb{R}^2$ is a path joining integer points via steps of length 1 either upward or rightward. How many lattice paths are there from $(0,0)$ to $(3,3)$?*

Exercise 3.3. *How many seven-digit phone numbers exist in which the digits are all nondecreasing?*

Exercise 3.4. *In how many ways can you roll a six-sided die three times and get different values each time?*

All of these are "counting" problems, in that they're asking you to figure out how many objects of a specific kind exist. However, because the sets in question are trickily defined, these problems are much harder than our "how many elements are in $\{3, 5, 7, \text{Snape}\}$ question.

To approach them, we'll need some new counting techniques! This is the goal of this chapter: we're going to study **combinatorics**, the art of counting, and develop techniques for solving problems like the ones above.

Let's start off with a simple task:

Problem 3.1. Suppose that we have 3 different postcards and 2 friends. In how many ways can we mail out all of our postcards to our friends while we're on vacation?

Answer. Let's give our cards names A, B, C, and also give our friends names X and Y, for easy reference.

In the setup above, a valid "way" to mail postcards to friends is some way to assign each postcard to a friend (because we're mailing out all three of our postcards.) To do this, think of going through each card A, B, C one-by-one and choosing a friend X, Y for each card.

By using brute-force, we can just enumerate all of the possibilities:

- X gets A, B, C, Y gets nothing.
- X gets A, B, Y gets C.
- X gets A, C, Y gets B.

- X gets B, C, Y gets A.
- X gets A, Y gets B, C.
- X gets B, Y gets A, C.
- X gets C, Y gets A, B.
- X gets nothing, Y gets A, B, C.

This works, and tells us that there are $\boxed{8}$ different ways to do this. However, if we had more cards this brute-force process seems like a bad idea:

Problem 3.2. Suppose that we have 10 different postcards and 4 friends. In how many ways can we mail out all of our postcards to our friends while we're on vacation?

Answer. We could try a brute-force approach like before. It would work, if you had enough time! However, if you start doing this you'll quickly find yourself bored out of your mind; there are *tons* of ways to do this, and just trying to list all of them is exhausting.

Instead, to do this, let's think about a **process** to generate ways to send out cards. That is: think about going through each card one-by-one and choosing a friend for each card:

$$\underbrace{\boxed{\text{4 possible friends}}}_{\text{card 1}}\ \underbrace{\boxed{\text{4 possible friends}}}_{\text{card 2}}\ \ldots\ \underbrace{\boxed{\text{4 possible friends}}}_{\text{card 10}}$$

When doing this process, we have 4 choices of friend for each card, and any combination of those choices will give us a valid way to send out cards. Therefore, we should have

$$\underbrace{4 \cdot 4 \cdot \ldots \cdot 4}_{10\ 4's} = \boxed{4^{10}}$$

total ways in which we can send out our cards.

This looks like an excellent answer to a counting problem, and also a much better method than our first approach!

You'll see this called the **multiplication principle** in other combinatorics resources online!

Alongside our answer, we also came up with a fairly interesting **method** for counting at the same time. Specifically, we had a set A of the following form:

1. Each element f of A could be constructed by making k choices in a row.
2. There was a fixed number n_i of possibilities for the i-th choice made in constructing any such element of A.
3. Therefore, we had $\underbrace{n_1 \cdot n_2 \cdot \ldots \cdot n_k}_{k\ n's}$ total elements in A.

By "fixed," we mean the following: the number of such choices is not affected by our other choices. That is, we'll always have n_i options for our i-th choice, no matter what our earlier choices actually were.

To get some practice with this principle, we study a number of examples of it in action:

Problem 3.3. How many binary strings of length 5 exist?

Answer. A binary string of length 5 has five characters in it, each of which is 0 or 1. Therefore, making such a string is the same thing as making five choices in a row, each of which has two options:

$$\underbrace{\boxed{0\ \text{or}\ 1}}_{\text{entry 1}}\ \underbrace{\boxed{0\ \text{or}\ 1}}_{\text{entry 2}}\ \underbrace{\boxed{0\ \text{or}\ 1}}_{\text{entry 3}}\ \underbrace{\boxed{0\ \text{or}\ 1}}_{\text{entry 4}}\ \underbrace{\boxed{0\ \text{or}\ 1}}_{\text{entry 5}}$$

By the multiplication principle, this can be done in $\boxed{2^5}$ ways.

Problem 3.4. Take a six-sided die, and roll it twice in a row. In how many ways can you do this and not see the same value repeat?

Answer. We can again think of this process as making two choices in a row:

$$\underbrace{\boxed{1,2,3,4,5 \text{ or } 6}}_{\text{roll 1}} \; \underbrace{\boxed{1,2,3,4,5 \text{ or } 6, \neq \text{ roll 1}}}_{\text{roll 2}}$$

There are 6 possibilities for the first roll. Once this is done, there are 5 possibilities for the second roll, as we can have any value show up other than the one that occurs in the first roll. Therefore, there are $6 \cdot 5 = \boxed{30}$ possibilities in total by the multiplication principle.

It is worth noting that the multiplication principle does not apply to all possible situations! Consider the following counting problem:

Problem 3.5. How many binary strings of length at most 3 exist?

Answer. Using the same logic as in Problem 3.3, we know that

- There are $2^3 = 8$ binary strings of length exactly 3,
- there are $2^2 = 4$ binary strings of length exactly 2, and
- there are $2^1 = 2$ binary strings of length exactly 1.

Also, we know that there is one binary string of length 0, namely the empty string λ.

Therefore, in total, there are $8 + 4 + 2 + 1 = \boxed{15}$ strings in total.

You'll sometimes see this "add disjoint things" process referred to as the **addition principle**.

Notice how at the end of the process above, we didn't multiply these numbers together! This is because the numbers $8, 4, 2, 1$ aren't all part of creating one giant mega-string. Instead, they each came from **disjoint** processes: that is, the strings of length 3 have no overlap with the strings of length 2, and so on/so forth. As such, we added them together!

To get some practice with spotting places where we use addition in counting, let's look at another problem:

Problem 3.6. Suppose that you have four friends Aang, Korra, Zuko, and Toph. In how many ways can you arrange them in a row, if Aang and Zuko are currently fighting and can't be placed next to each other?

Answer. If we just had the problem "In how many ways can we arrange Aang, Korra, Zuko, and Toph in a row," our problem is pretty straightforward! By using our multiple-choice framework from before,

$$\underbrace{\boxed{\text{A,K,Z,T}}}_{\text{friend 1}} \; \underbrace{\boxed{\text{A,K,Z,T}}}_{\text{friend 2}} \; \underbrace{\boxed{\text{A,K,Z,T}}}_{\text{friend 3}} \; \underbrace{\boxed{\text{A,K,Z,T}}}_{\text{friend 4}}$$

we can see that we have 4 choices for the first friend in our order, 3 for our second (as we can't repeat a friend), 2 for our third (can't choose any of the two previously-placed friends), and 1 for our last (everyone else is already in place!) So, in total, we have $4 \cdot 3 \cdot 2 \cdot 1 = 24$ ways to do this.

However, in this process we've allowed all of the possible arrangements, including ones like "A,Z,T,K" where Aang and Zuko are together. How do we correct for this?

Well, let's try to count all of the ways in which Aang and Zuko **could** be placed together! There are two ways in which this can happen:

- Aang and Zuko are placed in the order "AZ." To find all of the arrangements where this could happen, think of all of the ways to place three things $\boxed{\text{T}}$, $\boxed{\text{K}}$, $\boxed{\text{AZ}}$ in order! There are $3 \cdot 2 \cdot 1 = 6$ ways to do this, by the same reasoning as above.
- Aang and Zuko are placed in the order "ZA." To find all of the arrangements where this could happen, think of all of the ways to place three things $\boxed{\text{T}}$, $\boxed{\text{K}}$, $\boxed{\text{ZA}}$ in order! There are again $3{\cdot}2{\cdot}1 = 6$ ways to do this.

Because the "AZ" and "ZA" cases are completely distinct, we add them to get that there are $6 + 6 = 12$ ways for Aang and Zuko to be placed together in either order.

Finally, in any way of arranging our friends at all, we either have Aang and Zuko together or not. Therefore, we add these situations together, to get

$$\begin{aligned}\boxed{\text{all arrangements}} &= \boxed{\text{arrangements with Aang and Zuko together}} + \boxed{\text{arrangements with Aang and Zuko separate}}\\ 24 &= 12 + \boxed{\text{arrangements with Aang and Zuko separate}}.\end{aligned}$$

In other words, there are $24 - 12 = \boxed{12}$ arrangements where Aang and Zuko are separate!

One particularly useful thing we can do with counting arguments is **measure probability**. That is: suppose that we have an experiment in which all of the individual outcomes are

- **equally likely**, and
- **mutually exclusive**.

To give some examples of experiments like this:

- Suppose that you were rolling a fair 6-sided die. In this case, there are 6 outcomes: the die could show a 1,2,3,4,5, or 6. These outcomes are all equally likely if our die is fair: they're also all mutually exclusive, as a die can only show one number at a time.
- Suppose that you flipped a fair coin. In this case, there are 2 outcomes: heads or tails. They're equally likely if the coin is fair, and they're both mutually exclusive, as a coin cannot simultaneously be both heads and tails.
- Suppose that you have a standard 52-card deck, and reveal the top card. In this case, there are 52 possible outcomes, i.e. the 52 possible cards in the deck, and each of them are mutually exclusive (because you're only revealing one card.)
- Suppose that you have a bag containing 12 cookies, and you take one cookie out. In this case, there are 12 events (the 12 possible cookies you could have picked) and they're all mutually exclusive (because you are only taking one cookie.)

In situations like this, we have the following very useful observation:

Observation 3.5. *Take any experiment in which all of the individual outcomes are equally likely and mutually exclusive. Let n be the number of all possible outcomes. Then, the* ***probability*** *that any individual outcome happens when you run that experiment is* $\boxed{\frac{1}{n}}$.

More generally, take any set A of outcomes and let a denote the number of elements in A. Then the probability of running your experiment and having an outcome from A happening is $\boxed{\frac{a}{n}}$.

This is very useful! By using our newfound ability to count complicated objects, we can now determine the likelihood of various events happening.

This probably feels somewhat abstract, so let's illustrate this with a few examples:

Problem 3.7. Take two fair 6-sided dice and roll them. What is the probability that the sum of these dice is 7?

Answer. On one hand, we know that there are six possible ways for our dice to sum to 7: the six pairs $(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)$ each correspond for a way for to roll two dice and get a 7. (Notice that we count $(1,6)$ and $(6,1)$ differently! This is because our two dice are different, and so rolling a 1 on the first die and a a 6 on the second is a different event than rolling a 6 followed by a 1.)

On the other hand, we know that there are 36 possible outcomes for a given roll of our two dice; this is because there are 6 choices for what the first die can be, 6 for the second, and we multiply these together.

Thus, the odds that we roll two dice and they sum to 7 is $\frac{6}{36} = \frac{1}{6}$.

Problem 3.8. You and three of your friends are graduating. At graduation, you each throw your hats in the air to celebrate! At the end, you each pick up a hat from the ground. What are the odds that each of you pick up your own hat? (Assume that you're picking up hats at random, i.e. you're as likely to pick up your own hat as anyone else's hat: also assume that for some reason no-one else is graduating, and so there are only four possible hats to pick up.)

Answer. It takes a bit more thought to come up with the experiment structure here. To do so, let's label you and your friends 1,2,3 and 4, and similarly label the hats 1,2,3 and 4. If we do so, then we can describe any way for you and your friends to pick up hats by just listing the numbers 1,2,3,4 in some order: that is, $(2,1,3,4)$ would describe the situation where 1 has 2's hat, 2 has 1's hat, and 3 and 4 have their own hats.

In this setting, there are as many ways to pick up hats as there are ways to write the numbers 1,2,3,4 in some order. We know from our work earlier that there are 24 ways for this to happen in (it's the same thing as ordering our four friends in Problem 3.6!)

There is also only one way for everyone to get their own hat: namely, the ordering (1,2,3,4). Therefore the probability that everyone picks up their own hat is $\frac{1}{24}$.

3.2 Ordered Choice

A special case of the counting processes we studied earlier is the following:

Observation 3.6. ***(Ordered choice with repetition.)*** *Suppose that you are choosing k objects from a set of n things, where you care about the order in which you choose your objects and can repeatedly pick the same thing if desired. There are* $\boxed{n^k}$ *many ways to make such a choice.*

This principle is remarkably handy! We can use it to answer one of our problems from the start of this chapter:

Answer to Exercise 3.1. In this problem, we're trying to count the number of five-letter strings that are **palindromes** (i.e. can be read the same way forwards and backwards.)

To do this, start by noticing that any five-letter palindrome can be constructed by taking an arbitrary three-letter string and sticking its second and first characters at the end of the string: e.g. you can transform "rad" to "radar," "ten" to "tenet" and "eev" to "eevee." This process is clearly reversible: i.e. you can take any five-letter palindrome and cut off its last two letters to get a 3-letter string.

Any such three-letter string is formed by making three choices in a row, and we have 26 choices for each letter; this gives us $\boxed{26^3}$ many such strings, and thus 26^3 many five-letter palindromes.

This can get a bit more complex, however. Let's try changing our postcard problem a bit from before:

Problem 3.9. Suppose that we have k different kinds of postcards, n friends, and that we want to mail these postcards to our friends. Last time, however, it was possible that we just mailed all of our postcards to the same friend. That's a bit silly, so let's add in a new restriction: let's never send any friend more than one postcard.

In how many ways can we mail out postcards now?

Answer. We can still describe each way of sending postcards as a sequence of choices:

$$\underbrace{\boxed{\text{? choices}} \cdot \boxed{\text{? choices}} \cdot \ldots \cdot \boxed{\text{? choices}}}_{k \text{ total slots}}$$

As before, we still have n possibilities for who we can send our first card to. However, the "ordered choice with repetition" principle doesn't immediately apply here: because we don't want to repeat any of our friends, we only have $n-1$ choices for our second slot, instead of n as before! In general, we have the following sequence of choices:

A common question students ask about this calculation: why do we go to $n-1$ and not n in the product above? Well: we have k total slots. In the first slot, none of our choices were eliminated yet! In the second slot, however, we've eliminated one choice with our first slot. By the third slot we've eliminated two possibilities, by the fourth we've eliminated three possibilities, and in general in the i-th slot we've eliminated $i-1$ possibilities. This leaves us with $k-1$ possibilities eliminated by the time we get to the k-th slot!

$$\underbrace{\boxed{n \text{ choices}} \cdot \boxed{n-1 \text{ choices}} \cdot \boxed{n-2 \text{ choices}} \cdot \ldots \cdot \boxed{n-(k-1) \text{ choices}}}_{k \text{ total slots}},$$

which translates into

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-(k-1))$$

many choices in total.

We can simplify this expression considerably by introducing a useful function:

Definition 3.1. *For any nonnegative integer n, we define the **factorial** of n, written $n!$, as the following product:*

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$$

If $n = 0$, we think of this as the "empty product," and say that $0! = 1$.

With this notation in mind, we can simplify our answer to the postcard problem considerably: notice that

$$\begin{aligned} n \cdot (n-1) \cdot \ldots \cdot (n-(k-1)) &= \frac{\Big(n \cdot (n-1) \cdot \ldots \cdot (n-(k-1))\Big) \cdot \Big((n-k) \cdot (n-(k+1)) \cdot \ldots \cdot 3 \cdot 2 \cdot 1\Big)}{\Big((n-k) \cdot (n-(k+1)) \cdot \ldots \cdot 3 \cdot 2 \cdot 1\Big)} \\ &= \boxed{\frac{n!}{(n-k)!}}. \end{aligned}$$

We can use this idea to describe another counting method:

Observation 3.7. ***(Ordered choice without repetition.)*** *Suppose that you are choosing k objects from a set of n things, where you care about the order in which you choose your objects, but can only pick an object at most once. There are* $\boxed{\dfrac{n!}{(n-k)!}}$ *many ways to make such a choice if $k \leq n$, and 0 ways otherwise (as we'll run out of choices!)*

To practice using this observation, let's work a pair of problems:

Problem 3.10. If you take a six-sided die and roll it three times, what is the probability that you get different values each time?

Answer. If we keep track of what our first/second/third roll of our die is, then there are 6^3 many possible ways for us to roll a 6-sided die three times: we have 6 possible outcomes, we're choosing one of those outcomes 3 times, and the order matters + repeats are allowed.

Conversely, if we want to know the number of ways to roll a 6-sided die and have no repeated values, this is $\frac{6!}{3!} = 6 \cdot 5 \cdot 4$, by our order matters + no repeats process described above. (Note that this answers Exercise 3.4!)

Therefore, in total, we have a $\frac{6 \cdot 5 \cdot 4}{6^3} \approx 0.56 = \boxed{56\,\%}$ chance of this happening.

Problem 3.11. Suppose that you have six lightbulbs: two identical red bulbs, and one green / one blue / one white / one pink.

In how many ways can you screw these bulbs into a string of six lightbulb sockets? (Assume that the order in which we string these bulbs matter: i.e. we consider "$RRGBWP$" and "$PWBGRR$" to be different.)

Answer. First, we notice that if we could distinguish between our two red bulbs, this problem is a straightforward ordered choice without repetition problem: we would have 6 different bulbs, 6 sockets, and thus $6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 6!$ ways to do this.

However, these red bulbs are identical! This creates a bit of a problem: if we label our red bulbs R_1, R_2, then the method above thinks that "R_1WGR_2BP" and "R_2WGR_1BP" are different strings, even though to us they are identical.

How can we correct for this? Well, just notice the following: given any string where our red bulbs are not labeled, there are exactly two ways to label them: either label the leftmost red bulb "R_1" and the other "R_2", or vice-versa. In other words, there are twice as many strings where we have labelings as strings where we don't!

Therefore, we can just divide our earlier answer by 2 to get $\boxed{\dfrac{6!}{2}}$ as our final answer.

3.3 Unordered Choice

In the section above, we discussed how to choose things in situations where we both could and could not "repeat" our choice. In both scenarios, however, the order in which we made these choices mattered!

This does not always happen in real life. Consider the following example:

Problem 3.12. Suppose that we have just one friend that we want to send postcards to. We still have n different kinds of postcards, but now want to send that one friend k different postcards in a bundle (say as a gift!) In how many ways can we pick out a set of k cards to send our friend?

In mathematics: whenever you have a problem at hand, constantly look for modifications like these to make to the problem! If you're stuck, it can give you different avenues to approach or think about the problem; conversely, if you think you understand the problem, this can be a way to test and deepen that understanding.

In this problem, we have n different kinds of postcards, and we want to find out how many ways to send k different cards to a given friend. At first glance, you might think that this is the same as the answer to our second puzzle: i.e. we have k slots, and we clearly have n choices for the first slot, $n-1$ choices for the second slot, and so on/so forth until we have $n-(k-1)$ choices for our last slot.

This would certainly seem to indicate that there are $\dfrac{n!}{(n-k)!}$ many ways to assign cards. However, our situation from before is not quite the same as the one we have now! In particular: notice that the order in which we pick our postcards to send to this one friend does not matter to our friend, as they will receive them all at once anyways! Therefore, our process above is **over-counting** the total number of ways to send out postcards: it would think that sending card X and Y is a different action to sending card Y and card X!

To fix this, we need to **correct** for our over-counting errors above. Notice that for any given set of k distinct cards, there are $k!$ different ways to order that set: this is because in ordering a set of k things, you make k choices for where to place the first element, $k-1$ choices for where to place the 2nd element, and so on / so forth until you have just 1 choice for the k-th element.

Therefore, if we are looking at the collection of ordered length-k sequences of cards, each unordered sequence of k cards corresponds to $k!$ elements in this ordered sequence! That is, we have the following equality:

$$\boxed{\begin{array}{c}\text{Unordered ways to}\\ \text{pick } k \text{ cards}\\ \text{from } n \text{ choices}\end{array}} \cdot k! = \boxed{\begin{array}{c}\text{Ordered ways to}\\ \text{pick } k \text{ cards}\\ \text{from } n \text{ choices}\end{array}} = \frac{n!}{(n-k)!}$$

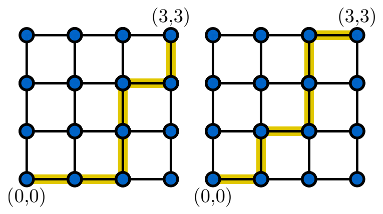
Therefore, if we want to only count the number of unordered ways to pick k cards from n choices, we can simply divide both sides of the above equation by $k!$, to get

$$\boxed{\begin{array}{c}\text{Unordered ways to}\\ \text{pick } k \text{ cards}\\ \text{from } n \text{ choices}\end{array}} = \frac{n!}{k!(n-k)!}$$

This concept — given a set of n things, in how many ways can we pick k of them, if we don't care about the order in which we pick those elements — is an incredibly useful one, and as such leads itself to the following definition:

Definition 3.2. ***(Unordered choice without repetition.)*** *The* ***binomial coefficient*** $\binom{n}{k}$ *is the number of ways to choose k things from n choices, if repeated choices are not allowed and the order of those choices does not matter.*

Observation 3.8. *By the working above, we can see that* $\displaystyle\binom{n}{k} = \frac{n!}{k!(n-k)!}$ *for any natural numbers k, n with $k \leq n$. For $k > n$, we have $\binom{n}{k} = 0$ by the same reasoning as before: we cannot choose more than n distinct things from a set of n possibilities!*



This observation lets us solve a few more problems:

Answer to Exercise 3.2. In this problem, we're trying to count the number of lattice paths from $(0,0)$ to $(3,3)$. How can we do this?

Well: notice that any path from $(0,0)$ to $(3,3)$ will need to take 6 steps; of those 6 steps, precisely 3 must be to the right and the remaining 3 must be upward. Therefore, we can create any such path by just "choosing" 3 out of our 6 steps to be the rightward steps!

There are $\boxed{\binom{6}{3}}$ many ways to choose such a set of rightward steps; this is because we don't repeat any of these choices (i.e. we have to pick three **different** places to go right), and because the order in which we decide which steps are rightward steps doesn't matter (i.e. picking steps 2, 3 and 5 to be rightward steps is the same as picking steps 5, 2 and 3 to be rightward steps.) Therefore, we have our answer: it's $\binom{6}{3} = \frac{6!}{3!3!} = \frac{6\cdot5\cdot4}{3\cdot2\cdot1} = \boxed{20}$.

Problem 3.13. Let's return to Problem 3.8, and tweak its setup a bit: what's the probability that **no-one** picks up their own hat?

Answer. Like in Problem 3.8, we still have the same universe of 24 possible ways for hats to be distributed. However, the set of situations in which no-one picks up their own hat is harder to measure!

One solution here could be to just list out via brute-force all of the possible ways to shuffle hats around without repeats. This could work; try it, if you want!

A second, clever trick (that would let us solve this problem for any number of friends) is the following:

$$\boxed{\begin{array}{c}\text{Ways in which for}\\\text{no-one gets}\\\text{their own hat}\end{array}} = \boxed{\begin{array}{c}\text{All ways}\\\text{to return hats}\end{array}} - \boxed{\begin{array}{c}\text{All ways to}\\\text{return hats where}\\\text{someone gets their own hat}\end{array}}$$

On one hand, this is kind of a dumb observation: we're just saying that the set of situations where our experiment succeeds is all of the possibilities where our experiment does not fail. On the other hand, though, this idea of "think about what happens where we do not succeed" has already been proven to be quite useful: both in Problem 3.6, and more generally when we've used the "suppose we're wrong" argument method in our first chapter! It's a good trick, and it will serve us well here.

In particular, suppose that we group our situations by **cases**, related to the number of people who get their own hats back:

- **4 hats back to original owners**: There's just $\boxed{1}$ way to do this, as noted in our previous problem.
- **3 hats back to original owners**: This is impossible! If three people get their own hat back, then there's only one hat left for the fourth person, i.e. their own hat. So there are $\boxed{0}$ ways for this to happen.
- **2 hats back to original owners**: To count this, first count the number of ways to pick the lucky two people who get their own hat back. This is $\binom{4}{2} = \frac{4!}{2!2!} = 6$, because we're picking 2 people to get their hats back from a set of 4, and order doesn't matter / we don't repeat people.

  With this done, there's exactly one way for these hats to go back to their owners in the desired fashion for each such pair: the two chosen people get their own hat, and the other two swap hats. Therefore, there are $\boxed{6}$ ways for this to happen in total.

- **1 hat back to its original owner**: We count this in the same way! There are 4 ways to pick a person to get their own hat back: just pick one of the four people.

  With this done, in how many ways can our remaining people shuffle hats? Well: of the three remaining people, the first can choose either of the other two people's hats. The person whose hat was **not** chosen then has no choice: to avoid taking their own hat, they must take the first person's hat. This leaves the third person with no choice as well. Therefore, there are $2 \cdot 1 \cdot 1$ ways for this to happen for a given chosen person.

  Across our four possible ways to choose people, then, there are $\boxed{2 \cdot 4 = 8}$ many ways for this to happen.

Therefore, the number of ways in which people get their own hats is $1 + 6 + 8 = 15$, and so the number of ways in which this does not happen is $24 - 15 = 9$. Thus, our probability is $\frac{9}{24} = \frac{3}{8}$.

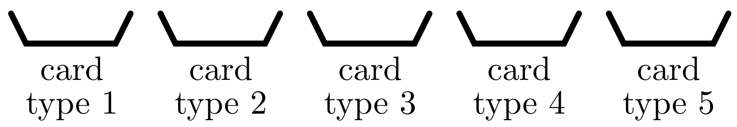
To illustrate one last counting process, we return to our postcard problem:

Problem 3.14. Suppose that we are at the shops and want to buy a bunch of postcards to send out to our friends. The shop sells n different kinds of postcards, and has tons of each kind. We want to buy k cards (possibly with repetitions, if there's a specific card design we like and want to send to many people.) In how many ways can we pick out a set of k cards to buy?

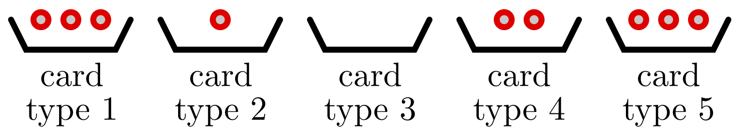
It first bears noting that this problem does not fall under the situations of our earlier problems. In this problem, our choices are unordered: i.e. we're just picking out a bundle of cards to buy, and the order in which they're bought is irrelevant. Therefore, we cannot use the "ordered choice with repetitions" observation we made earlier, as this would massively overcount things (i.e. we'd count orders of the same cards as different if the cashier rang things up in a different order, which is silly.)

However, unlike our two "ordered/unordered choice without repeats" situations, we can repeat choices! This means that this is not at all like those situations: in particular, k can be larger than n and we will still have lots of possibilities here, whereas in in the "without repeats" situations this was always impossible. So we need a new method!

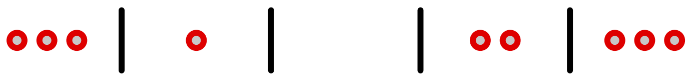
To develop this method, think of the n different kinds of postcards as n "bins." Here's a visualization for when $n = 5$:



Picking out k cards to buy, then, can be thought of as pulling a few cards from the first bin, a few from the second, and so on/so forth until we've pulled out k cards in total. In other words, this is the **same problem** as distributing k balls amongst n bins:

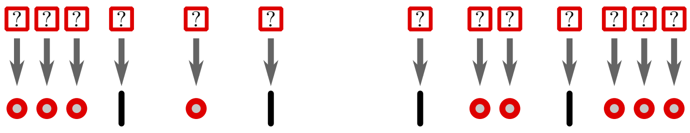


To do *this* task, replace the n bins with $n - 1$ "dividers" between our choices. This separates our choices just as well as the bins did, so this is still the same problem.



Now, **forget the difference between objects and dividers**! That is, take the diagram above and suppose that you cannot tell the difference between an object and a divider between our choices.

How can we return this back to a way to choose k things from n choices? Well: take the set of $k + (n - 1)$ objects, of which k used to be things and $n - 1$ were dividers. Now choose $n - 1$ of them to be dividers! This returns us back to a way to pick out k things from n choices.



In particular, note that given any set of $k + (n - 1)$ placeholders, we can turn it into a way to choose k things from n choices with repetition by performing such a choice! Therefore, there are as many ways to make such choices as there are ways to choose $n - 1$ things from a set of $k + (n - 1)$ options to be placeholders. This second choice is unordered and without repetition (we want all of the placeholders to be different, and don't care about the order in which we pick the placeholders: just the elements that are chosen!) Therefore, we can use our "unordered choice without repetition" principle to see that there are $\binom{k+n-1}{n-1}$ many ways to do this!

In other words, we have the following observation:

Observation 3.9. ***(Unordered choice with repetition.)*** *The number of ways to choose k things from n choices, where we do not care about the order in which we make our choices but allow choices to be repeated, is $\binom{k+n-1}{n-1}$.*

To practice this, let's answer our last two problems of this chapter:

Problem 3.15. Suppose that you have ten identical cookies, and want to distribute them to four of our friends, so that each friend gets at least one cookie. In how many ways can we do this?

Answer. First, if every friend gets at least one cookie, we can start by just distributing one cookie to each friend! This leaves us with 6 cookies left over to distribute further to our friends.

If we think of taking each cookie and "choosing" a friend to give it to, this is unordered choice with repetition: the order doesn't matter because the cookies are identical, and repeats are allowed because we can give one friend multiple cookies.

By our formula above, then, there are $\binom{6+4-1}{4-1} = \binom{9}{3} = \frac{9 \cdot 8 \cdot 7}{3 \cdot 2} = \boxed{84}$ ways for this to happen.

Answer to Exercise 3.3. In this problem, we're trying to determine how many seven-digit phone numbers exist, in which the digits are nondecreasing? (By "nondecreasing" here, we just mean that each digit is at least as big to the digit to its left; i.e. 122-2559 is a valid phone number, but 321-1234 would not be.)

With this understood, we claim that our problem can be reduced to an "unordered choice with repetition" task as follows: consider any way to choose seven numbers from the set of digits $\{0, 1, \ldots 9\}$, without caring about the order and with repetition allowed.

On one hand, we claim that any such choice can be turned into a nondecreasing phone number! Just list the digits here in order of their size; i.e. if you picked three 1's, a 2, a 3, and two 5's, write down 111-2355. This process also clearly generates any such phone number (just pick its digits!), and so the number of seven-digit nondecreasing phone numbers is just the number of unordered ways to choose seven things from the set of digits $\{0, 1, \ldots 9\}$ with repetition.

By our above formula, there are $\binom{7+10-1}{10-1} = \binom{16}{9} = 11440$ many such numbers. Success!

3.4 Practice Problems

1. (-) How many binary strings of length 10 exist?
2. How many three-digit numbers exist where all of the digits are different? (Hint: the answer is **not** $10 \cdot 9 \cdot 8 \ldots$)
3. You have ten indistinguishable cookies that you've baked for four of your friends Jianbei, Sione, Sina and Julia. You want to give away all ten of your cookies to your friends, and for each friend to get at least one cookie. You also know that Sione wants an even number of cookies, so he can share them with a friend.

   In how many ways can you give away your cookies?
4. You're a Pokémon master! You have collected exactly one of all 151 Pokémon in the base game. You want to assemble a team to take on the Elite Four: doing this involves choosing a team of 6 Pokémon (in which the order does not matter) and then designating one of those six to be the "lead" on your team (i.e. the first one to go out.) In how many ways can you do this?
5. For any two natural numbers $n, k \in \mathbb{N}$, let $C(n, k)$ denote the number of ways to choose k **unordered** objects from a set of n distinct objects without repeats, and let $P(n, k)$ denote the number of ways to choose k **ordered** objects from a set of n distinct objects without repeats.

   For what values of n, k is $C(n, k) = P(n, k)$?
6. (-) Take the collection of all length-10 binary strings and pick one at random. What are the odds that the sum of all digits in your string is even? Does this change if you were looking at length-11 binary strings?
7. Take a standard 52-card deck of playing cards, shuffle it, and draw a hand of five cards. What is the probability that your hand has a three-of-a-kind?
8. Can you find three events A, B, C such that
   - the probability of A and B happening at the same time is $\frac{1}{2}$,
   - the probability of A and C happening at the same time is $\frac{1}{2}$,
   - the probability of B and C happening at the same time is $\frac{1}{2}$, and yet somehow
   - the probability that all **three** of A, B, C happening at the same time is 0?

9. (+) Suppose that there's an election! Two candidates, Sherlock and Moriarty, are running for office. Suppose that Sherlock receives 8 votes and Moriarty receives 7 votes, and that these votes are being counted up one-by-one to create a running total.

   What is the probability that Sherlock is never behind in this running total? In general, if Sherlock got s votes and Moriarty got m votes, what is this probability?

10. (+) Auckland has about 1 million residents. Suppose each resident has a jar with 100 coins in it.

    Two jars are considered to be "equivalent" if they have the same number of 10c, 20c, 50c, \$1 and \$2 coins in them.

    How many different (i.e. nonequivalent) jars of coins exist? Is it theoretically possible that every person in Auckland has a different jar of coins?

Exercise 4.1. *Two processes that you can use to multiply two nonnegative integers a, b together are listed below:*

Algorithm 4.1.

1. *Define a new number prod, and initialize it (i.e. set it equal) to 0.*
2. *If $\mathbf{a} = 0$, stop, and return the number prod.*
3. *Otherwise, add b to prod, and subtract 1 from a. Then go to 2.*

Algorithm 4.2.

1. *Define a new number prod, and initialize it (i.e. set it equal) to 0.*
2. *If $\mathbf{a} = 0$, stop, and return the number prod.*
3. *Otherwise, if a is odd subtract 1 from a and set $prod = prod + b$.*
4. *Divide a by 2, and multiply b by 2.*
5. *Go to step 2.*

In general, which of these is the faster way to multiply two numbers? Why?

An example run of Algorithm 4.1 when $a = 3, b = 12$:

step	a	b	$prod$
1	3	12	0
2			
3	2		12
2			
3	1		24
2			
3	0		$\boxed{36}$
2	(halt!)		

Similarly, an example run of Algorithm 4.2 when $a = 3, b = 12$:

step	a	b	$prod$
1	3	12	0
2			
3	2		12
4	1	24	
2			
3	0		$\boxed{36}$
4	0	48	
2	(halt!)		

4.1 Functions in General

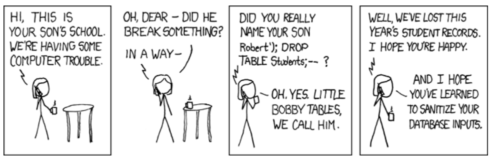
In your high-school mathematics classes, you've likely seen functions described as things like "$f(x) = 2x+3$" or "$g(x) = \max(x, y)$." When we're writing code, however, we don't do this! That is: in most programming languages, you can't just type in expressions like the ones before and trust that the computer will understand what you mean. Instead, you'll often write something like the following:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Notice how in this example we didn't just get to define the rules for our function: we also had to specify the kind of **inputs** the function should expect, and also the type of **outputs** the function will generate! On one hand, this creates more work for us at the start: we can't just tell people our rules, and we'll often find ourselves having to go back and edit our functions as we learn what sorts of inputs we actually want to generate.

On the other hand, though, this lets us describe a much broader class of functions than what we could do before! Under our old high-school definition for a function, we just assumed that functions took in numbers and returned numbers. With the above idea, though, we can have functions take in and output anything: multiple numbers, arrays, text files, whatever!

On the third(?) hand, enforcing certain restrictions on the types of inputs and outputs to a function is also a much more **secure** way to think about functions in computer science. If you're writing code in a real-life situation, you should always expect malicious or just clueless users to try to input the worst possible data to your functions. As a result, you can't just have a function defined by the rule $f(x) = \frac{1}{x}$ and trust that your users out of the goodness of their hearts will never input 0 just to see what happens! They'll do it immediately (as well as lots of other horrifying inputs, like 🐤, $\frac{1}{0}$, $1 - 0.\overline{9}$, ...) just to see what happens.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
https://xkcd.com/327/

This is why many programming languages enforce type systems: i.e. rules around their functions that specifically force you to declare the kinds of inputs and outputs ahead of time, like we've done above! Doing this is an important part of writing bug-free and secure code.

As this is a computer science class, we should have a definition of function that matches this concept. We provide this here:

Definition 4.1. *Formally, a* ***function*** *consists of three parts:*

- *A collection A of possible* ***inputs****. We call this the* ***domain*** *of our function.*
- *A collection B describing the type of* ***outputs*** *that our function will generate. We call this the* ***codomain*** *of our function.*
- *A rule f that takes in inputs from A and generates outputs in B.*

Furthermore, in order for this all to be a function, we need it to satisfy the following property:

For every potential input a from A, there should be exactly one b in B such that $f(a) = b$.

In other words, we never have a value a in A for which $f(a)$ is undefined, as that would cause our programs to crash! As well, we also do not allow for a value $a \in A$ to generate "multiple" outputs; i.e. we want to be able to rely on $f(a)$ not changing on us without warning, if we keep a the same.

Example 4.1. Typically, to define a function we'll write something like "Consider the function $f : \mathbb{Z} \to \mathbb{Q}$, defined by the rule $f(n) = \frac{1}{n^2+1}$. This definition tells you three things: what the domain is (the set the arrow starts from, which is the integers in this case), what the codomain is (the set the arrow points to, which is the rational numbers in this case), and the rule used to define f.

Example 4.2. $f : \mathbb{Z} \to \mathbb{Z}$ defined by the rule "$f(x) = y$ if and only if $x = y^2$" is not a function. There are many reasons for this:

- There are values in the domain that do not get mapped to any values in the codomain by our rule. For instance, consider $x = -1 \in \mathbb{Z}$. There is no value $y \in \mathbb{R}$ such that $-1 = y^2$, because no integer when squared is negative! Therefore, x is not mapped to any value y in the codomain, and so we do not regard f as a function.

- There are also values in the domain that get mapped to multiple values in the codomain by our rule. For instance, consider $x = 1 \in \mathbb{Z}$. Because $1 = y^2$ has the two solutions $y = \pm 1$, this rule maps $x = 1$ to the two values $y = \pm 1$. This is another reason why f is not a function!

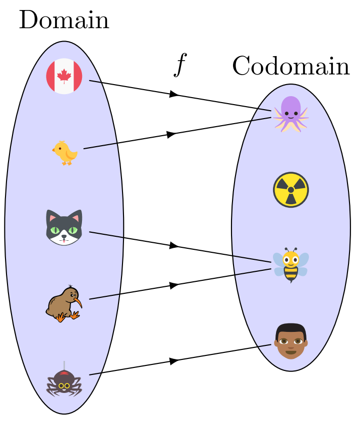
Example 4.3. Let A be the set of all students at the University of Auckland, and B be the set of all integers. We can define a function $f : A \to B$ by defining $f(a)$ to be equal to that student's current ID number. This is a function, because each student has a unique ID number!

However, if we tried to define a function $g : B \to A$ by the rule $g(b)$ = the student whose ID number is b, we would fail! This is because there are many integers that are not ID numbers: for example, no student has ID number -1, or 10^{100}.

While the objects above have had relatively "nice" rules, not all functions can be described quite so cleanly! Consider the following example:

Example 4.4. Let A be the collection {🇨🇦, 🐤, 🐱, 🥝} of the Canada, cat, bird, and kiwi emojis, and let B be the collection {🐙, ☢️, 🐝, 👨🏾} of the octopus, radiation, bee, and man emojis. Consider $f : A \to B$, defined by the rules

$f($🇨🇦$) =$ 🐙, $f($🐤$) =$ 🐙, $f($🐱$) =$ 🐝, $f($🥝$) =$ 🐝, $f($🕷️$) =$ 👨🏾



This is a function, because we have given an output for every possible input, and also never sends an input to multiple different outputs. It's not a function with a simple algebraic rule like "$x^2 + 2x - 1$", but that's OK!

A useful way to visualize functions defined in this piece-by-piece fashion is with a diagram: draw the domain at left, the codomain at right, and draw an arrow from each x in the domain to its corresponding element $f(x)$ in the codomain.

Alongside the domain/codomain ideas above, another useful idea here is the concept of **range**:

Definition 4.2. *Take any function $f : A \to B$. We define the **range** of f as the set of all values in the codomain that our function **actually** sends values in the domain to. In other words, the range of f is the following set:*

$$\{b \in B \mid \text{there is some } a \in A \text{ such that } f(a) = b\}$$

Note that the range is usually different to the codomain! In the examples we studied earlier, we saw the following:

- $f : \mathbb{Z} \to \mathbb{Q}$ defined by the rule $f(n) = \dfrac{1}{n^2+1}$ does not output every rational number! Amongst other values, it will never output any number greater than 1 (as $\dfrac{1}{n^2+1} \le \dfrac{1}{0^2+1} = 1$ for every integer n.) As such, its codomain ($\mathbb{Q}$) is not equal to its range.
- The function $f : A \to B$ from Example 4.3, that takes in any student at the University of Auckland and outputs their student ID, does not output every integer: amongst other values, it will never output a negative integer! As such, its codomain (B) is not equal to its range.
- The emoji function in Example 4.4 never outputs ☢️, even though it's in the codomain.

Intuitively: we think of the codomain as letting us know what *type* of outputs we should expect. That is: in both mathematics or computer science, often just knowing that the output is "an integer" or "a binary string of length at most 20" or "a Unicode character" is enough for our compiler to work. As such, it's often much faster to just describe the type of possible outputs, instead of laboriously finding out exactly what outputs are possible!

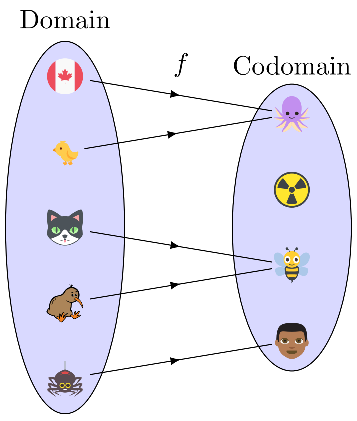
However, in some cases we *will* want to know precisely what values we get as outputs, and in that situation we will want to find the actual outputs: i.e. the **range**. To illustrate this, let's consider a few examples:

Example 4.5. Consider the function $f : \mathbb{Z} \to \mathbb{Z}$ given by the rule $f(x) = 2|x| + 2$. This function has range equal to all even numbers that are at least 2.

To see why, simply notice that for any integer x, $|x|$ is the "absolute value" of x: i.e. it's x if we remove its sign so that it's always nonnegative. As a result, $2|x|$ is always a nonnegative even number, and this $2|x| + 2$ must be a nonnegative even number that's at least 2.

That tells us that the only possible outputs are even numbers that are at least 2! However, we still don't know that all of those outputs are ones that we actually can get as outputs.

To see why this is true: take any even number that is at least 2. By definition, we can write this as $2k$, for some $k \geq 1$. Rewrite this as $2(k-1)+2$; if we do so, then we can see that $f(k-1) = 2|k-1|+2 = 2(k-1)+2$ (because if $k \geq 1$, then $k-1 \geq 0$ and so $|k-1| = k-1$.) As a result, we've shown that $f(k-1) = 2k$ for any $k \geq 1$, and thus that we can actually *get* any even number that's at least 2 as an output.

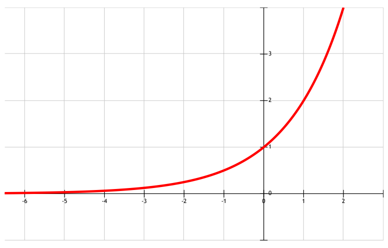


Example 4.6. Consider the emoji function from Example 4.4. If we look at the diagram we drew before, we can see that our function generates three possible outputs: 🐙, 🐝 and 👨🏾. Therefore, the collection of these three emojis is our range!

Example 4.7. Let A be the collection of all pairs of words in the English language, and B be the two values {true, false}. Define the function $f : A \to B$ by saying that $f(w_1, w_2)$ = true if the words w_1, w_2 rhyme, and false otherwise. For example, f(cat, bat) = true, while f(cat, cataclysm) = is false.

The range of this function is {true, false}, i.e. the same as its codomain! It is possible for the range and codomain to agree. (If this happens, we call such a function a **surjective** function. We're not going to focus on these functions here, but you'll see more about them in courses like Compsci 225 and Maths 120/130!)

Example 4.8. Let $\mathbb{R}$ denote the set of all real numbers (i.e. all numbers regardless of whether they're rational or irrational; alternately, anything you can describe with a decimal expansion.) Define the function $f : \mathbb{R} \to \mathbb{R}$ by the rule $f(x) = 2^x$.



This function has range equal to the set of all positive numbers! This takes more math to see than we currently have: again, take things like Maths 130 to see the "why" behind this. However, if you draw a graph of 2^x you'll see that the outputs (i.e. y-values) range over all of the possible positive numbers, as claimed.

To close this section, we give a useful bit of notation for talking about functions defined in terms of other functions: **function composition**.

Fact 4.2. *Given any two functions $f : B \to C, g : A \to B$, we can combine these functions via* ***function composition****: that is, we can define the function $f \circ g : A \to C$, defined by the rule $f \circ g(x) = f(g(x))$. We pronounce the small open circle symbol $\circ$ as "composed with."*

Example 4.9.

- If $f, g : \mathbb{R} \to \mathbb{R}$ are defined by the rules $f(x) = x + 1$ and $g(x) = x^2 - 1$, then we would have $g \circ f(x) = g(f(x)) = g(x+1) = (x+1)^2 - 1 = x^2 + 2x$.

- Notice that this is different to $f \circ g(x) = f(g(x)) = f(x^2-1) = (x^2-1)+1 = x^2$!

  In general, $f \circ g$ and $g \circ f$ are usually different functions: make sure to be careful with the order in which you compose functions.
- If $f, g : \mathbb{R} \to \mathbb{R}$ are defined by the rules $f(x) = 3x-1$ and $g(x) = \frac{x+1}{3}$, then $f \circ g(x) = f(g(x)) = f\left(\frac{x+1}{3}\right) = 3\frac{x+1}{3} - 1 = (x+1)-1 = x$.
- If $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}$ are defined by the rules $f(x) = 2^{x^2+1}$ and $g(x) = \log_2(x) - 1$, then $g \circ f(x) = g(f(x)) = g\left(2^{x^2+1}\right) = \log_2\left(2^{x^2+1}\right) - 1 = (x^2+1) - 1 = x^2$.

Here, $\mathbb{R}^+$ denotes the set of all positive real numbers.

Handy!

Notice that in the definition above, we required that the domain of f was the codomain of g. That is: if we wanted to study $f \circ g(x) = f(g(x))$, we needed to ensure that every output of g is a valid input to f.

This makes sense! If you tried to compose functions whose domains and codomains did *not* match up in this fashion, you'd get nonsense / crashes when the inner function g returns an output at which the outer function is undefined. For example:

Example 4.10.

- If $f : \mathbb{R} \setminus \{0\} \to \mathbb{R}$ is defined by the rule $f(x) = \frac{1}{x}$ and $g : \mathbb{R} \to \mathbb{R}$ is defined by the rule $g(x) = x^2 - 1$, then you might think that $f \circ g(x) = f(g(x)) = \frac{1}{x^2-1}$.

  However, this is not a function! When $x = \pm 1$, for example, we have $f(g(\pm 1)) = \frac{1}{(\pm 1)^2 - 1} = \frac{1}{0}$, which is undefined. This is why we insist that the codomain of g is the domain of f; we need all of g's outputs to be valid inputs to f.
- Let A be the set of all people in your tutorial room, B be the set of all ID numbers of UoA students, and C be the set of all ID numbers of Compsci 120 students. Then $f : C \to \mathbb{R}$ defined by taking any Compsci 120 student's ID and outputting their grade on the mid-sem test is a function; as well, $g : A \to B$, defined by mapping each person in your tutorial room to their ID number is a function.

  However, $f \circ g$, the function that tries to take each person in your tutorial room and output their mid-sem test score, is undefined! In particular, your tutor is someone in your tutorial room, who even though they do have an ID number, will not have a score on the mid-sem test. Another reason to insist that the codomain of g is the domain of f!

4.2 Algorithms

In the previous section, we came up with a "general" concept for function that we claimed would be better for our needs in computer science, as it would let us think of things like

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
```

```
    else
       result = num2;

    return result;
 }
```

as a function. However, most of the examples we studied in this chapter didn't feel too much like the code above: they were either fairly mathematical in nature (i.e. $f(n) = \frac{1}{n^2+1}$) or word-problem-oriented (i.e. the function that sent UoA students to their ID numbers.)

To fix this issue, this section will focus on the idea of an **algorithm**: that is, a way of describing in general a step-by-step problem-solving process that we can easily turn into code.

We start by describing what an algorithm is:

Definition 4.3. *An **algorithm** is a precise and unambiguous set of instructions.*

Typically, people think of algorithms as a set of instructions for solving some problem; when they do so, they typically have some restrictions in mind for the kinds of instructions they consider to be valid. For example, consider the following algorithm for proving the Riemann hypothesis:

1. Prove the Riemann hypothesis.
2. Rejoice!

On one hand, this is a fairly precise and unambiguous set of instructions: step 1 has us come up with a proof of the Riemann hypothesis, and step 2 tells us to rejoice.

On the other hand: this is not a terribly useful algorithm. In particular, its steps are in some sense "too big" to be of any use: they reduce the problem of proving the Riemann hypothesis to . . . proving the Riemann hypothesis. Typically, we'll want to limit the steps in our algorithms to **simple**, mechanically-reproducible steps: i.e. operations that a computer could easily perform, or operations that a person could do with relatively minimal training.

In practice, the definition of "simple" depends on the context in which you are creating your algorithm. Consider the algorithm for making delicious pancakes, given at right. This algorithm's notion of "simple" is someone who is (1) able to measure out quantities of various foods, and (2) knows the meaning of various culinary operations like "whisk" and "flip." If we wanted, we could make an algorithm that includes additional steps that define "whisking" and "flipping". That is: at each step where we told someone to whisk the flour, we could instead have given them the following set of instructions:

(a) Grab a whisk. If you do not know what a whisk is, go to this Wikipedia article and grab the closest thing to a whisk that you can find. A fork will work if it is all that you can find.

(b) Insert the whisk into the object you are whisking.

(c) Move the whisk around through the object you are whisking in counterclockwise circles of varying diameter, in such a fashion to mix together the contents of the whisked object.

In this sense, we can extend our earlier algorithm to reflect a different notion of "simple," where we no longer assume that our person knows how to whisk things. It still describes the same sets of steps, and in this sense is still the "same" algorithm – it just has more detail now!

An algorithm for pancakes!

1. Acquire and measure out the following ingredients:
   - 2 cups of buttermilk, or 1.5 cups milk + .5 cups yoghurt whisked together.
   - 2 cups of flour.
   - 2 tablespoons of sugar.
   - 2 teaspoons of baking powder.
   - 1/2 teaspoon of baking soda.
   - 1/2 teaspoon of salt.
   - 1 large egg.
   - 3 tablespoons butter.
   - Additional butter.
   - Maple syrup.
2. Whisk the flour, sugar, baking powder, baking soda, and salt in a medium bowl.
3. Melt the 3 tablespoons of butter.
4. Whisk the egg and melted butter into the milk until combined.
5. Pour the milk mixture into the dry ingredients, and whisk until just combined (a few lumps should remain.)
6. Heat a nonstick griddle/frypan on medium heat until hot; grease with a teaspoon or two of butter.
7. Pour 1/4 cup of batter onto the skillet. Repeat in various disjoint places until there is no spare room on the skillet. Leave gaps of 1cm between pancakes.
8. Cook until large bubbles form and the edges set (i.e. turn a slightly darker color and are no longer liquid,) about 2 minutes.
9. Using a spatula, flip pancakes, and cook a little less than 2 minutes longer, until golden brown.
10. If there is still unused batter, go to 5; else, top pancakes with maple syrup and butter, and eat.

This is a good recipe. Use it!

This concept of "adding" or "removing" detail from an algorithm isn't something that will always work; some algorithms will simply demand steps that cannot be implemented on some systems. For example, no matter how many times you type "sudo apt-get 2 cups of flour," your laptop isn't going to be able to implement our above pancake algorithm. As well, there may be times where a step that was previously considered "simple" becomes hideously complex on the system you're trying to implement it on!

We're not going to worry too much about the precise definition of "simple" in this class, because we're not writing any code here (and so our notion of "simple" isn't one we can precisely nail down) — these are the details we'll leave for your more coding-intensive courses.

Instead, let's just look at a few examples of algorithms! We've already seen one in this class, when we defined the % operation:

Algorithm 4.3. *This algorithm takes in any two integers a, n, where $n > 0$. It then calculates $a \% n$ as follows:*

- *If $a \geq n$, we repeatedly **subtract** n from a until $a < n$, and return the end result.*
- *If $a < 0$, repeatedly **add** n to a until $a > 0$, and return the end result.*
- *If neither of these cases apply, then we just return a.*

Second, we can turn Claim 1.6 into an algorithm for how to tell if a number is prime:

Algorithm 4.4. *This is an algorithm that takes in a positive integer n, and determines whether or not n is prime. It proceeds as follows:*

- *If $n = 1$, stop: n is not prime.*
- *Otherwise, if $n > 1$, find all of the numbers $2, 3, 4, \ldots \lfloor \sqrt{n} \rfloor$. Take each of these numbers, and test whether they divide n.*
- *If one of them does, then n is not prime!*
- *Otherwise, if none of them divide n, then by Claim 1.6, n is prime.*

This is a step-by-step process that tells us if a number is a prime or not! Notice that the algorithm itself didn't need to contain a proof of Claim 1.6; it just has to give us instructions for how to complete a task, and not justify *why* those instructions will work. It is good form to provide such a justification where possible, as it will help others understand your code! However, it is worth noting that such a justification is separate from the algorithm itself: it is quite possible (and indeed, all too easy) to write something that works even though you don't necessarily understand why.

For a third example, let's consider an algorithm to **sort** a list:

Algorithm 4.5. *The following algorithm, `InsertionSort`(L), takes in a list $L = (l_1, l_2, \ldots l_n)$ of n numbers and orders it from least to greatest. For example, `InsertionSort`$(1, 7, 1, 0)$ is $(0, 1, 1, 7)$. It does this by using the following algorithm:*

1. *If L contains at most one number, L is trivially sorted! In this situation, stop.*
2. *Otherwise, L contains at least two numbers. Let $L = (l_1, l_2, \ldots l_n)$, where $n \geq 2$. Define a pair of values $\mathtt{val}_{min}$, $\mathtt{loc}_{min}$, and set them equal to the value and location of the first element in our list.*
3. *One by one, starting with the second entry in our list and working our way through our entire list L, compare the value stored in $\mathtt{val}_{min}$ to the current value l_k that we're examining.*

   (a) *If $\mathtt{val}_{min} > l_k$, update $\mathtt{val}_{min}$ to be equal to l_k, and update $\mathtt{loc}_{min}$ to be equal to k.*
   (b) *Otherwise, just go on to the next value.*

4. *At the end of this process, $\mathtt{val}_{min}$ and $\mathtt{loc}_{min}$ describes the value and the location of the smallest element in our list. Swap the first value in our list with $l_{\mathtt{loc}_{min}}$ in our list: this makes the first value in our list the smallest element in our list.*
5. *To finish, set the first element of our list aside and run* `InsertionSort` *on the rest of our list.*

Back in our first chapter, to understand our two previous algorithms 4.3 and 4.4 we started by running these algorithms on a few example inputs! In general, this is a good tactic to use when studying algorithms; actually plugging in some concrete inputs can make othewise-obscure instructions much simpler to understand.

To do so here, let's run our algorithm on the list $(1, 7, 1, 0)$, following each step as written:

list	step	$\mathtt{loc}_{\mathtt{min}}$	$\mathtt{val}_{\mathtt{min}}$	current k	current l_k
(1,7,1,0)	1				
(1,7,1,0)	2	1	1		
(1,7,1,0)	3			2	7
(1,7,1,0)	3			3	1
(1,7,1,0)	3			4	0
(1,7,1,0)	3(a)	4	0		
(0,7,1,1)	4				
(0, $\boxed{7,1,1}$)	5				
(0, $\boxed{7,1,1}$)	1				
(0, $\boxed{7,1,1}$)	2	2	7		
(0, $\boxed{7,1,1}$)	3			3	1
(0, $\boxed{7,1,1}$)	3(a)	3	1		
(0, $\boxed{7,1,1}$)	3			4	1
(0, $\boxed{1,7,1}$)	4				
(0,1, $\boxed{7,1}$)	5				
(0,1, $\boxed{7,1}$)	1				
(0,1, $\boxed{7,1}$)	2	3	7		
(0,1, $\boxed{7,1}$)	3			4	1
(0,1, $\boxed{7,1}$)	3(a)	4	1		
(0,1, $\boxed{1,7}$)	4				
(0,1,1, $\boxed{7}$)	5				
(0,1,1, $\boxed{7}$)	1				

Here, we use the $\boxed{7, 1, 1}$, $\boxed{7, 1}$ and $\boxed{7}$ boxes to visualize the "rest of our list" part of step 5 in our algorithm.

This worked! Moreover, doing this by hand can help us see an argument for *why* this algorithm works:

Claim 4.1. `InsertionSort` *(i.e. algorithm 4.5) works.*

Proof. In general, there are three things we need to check to show that a given algorithm works:

- **The algorithm doesn't have any bugs**: i.e. every step of the process is defined, you don't have any division by zero things or undefined cases, or stuff like that.

  This is true here! The only steps we perform in this algorithm are comparisons and swaps, and the only case we encounter is "$\mathtt{val}_{\mathtt{min}} > l_k$ is true" or "$\mathtt{val}_{\mathtt{min}} > l_k$ is false," which clearly covers all possible situations. As such, there are no undefined cases or undefined operations.

- **The algorithm doesn't run forever**: i.e. given a finite input, the algorithm will eventually stop and not enter an infinite loop.

  This is also true here! To see why, let's track the number of comparisons and write operations performed by this process, given a list of length n as input:

  – Step 1: one comparisons (we checked the size of the list.)
  – Step 2: two write operations (we defined $\mathtt{val}_{\mathtt{min}}$, $\mathtt{loc}_{\mathtt{min}}$.)

  - Step 3: $(n-1)$ comparisons and possibly $2(n-1)$ write operations.

    To see why: note that for all of the $n-1$ entries in our list from l_2 onwards, we looked up the value in l_k and compared it to the value we have in $\texttt{val}_{\texttt{min}}$, which gives us $n-1$ comparisons. If it was smaller, we rewrote the values in $\texttt{val}_{\texttt{min}}$ and $\texttt{loc}_{\texttt{min}}$ in 3(a).
  - Step 4: 2 write operations (to swap these values.)
  - Step 5: 1 write operation (to resize the list to set the first element aside), and however many operations we need to sort a list of $n-1$ numbers with `InsertionSort`.

In total, then, we have the following formula: if $\texttt{InsertionSortSteps}(n)$ denotes the maximum number of operations in total needed to sort a list with our algorithm, then

$$\begin{aligned}\texttt{InsertionSortSteps}(n) &= 1+2+(n-1)+2(n-1)+2+1+\texttt{InsertionSortSteps}(n-1)\\ &= 3n+3+\texttt{InsertionSortSteps}(n-1).\\ &= 3(n+1)+\texttt{InsertionSortSteps}(n-1).\end{aligned}$$

At first, this looks scary: our function is defined in terms of itself! In practice, though, this is fine. We know that $\texttt{InsertionSortSteps}(1) = 1$, because step 1 immediately ends our program if the list has size 1.

Therefore, our formula above tells us that if we set $n = 2$, we have

$$\texttt{InsertionSortSteps}(2) = (3\cdot(2+1)) + \texttt{InsertionSortSteps}(2-1) = 9+1 = \boxed{10},$$

by using our previously-determined value for $\texttt{InsertionSortSteps}(1)$. Still finite!

We can do this again for $n = 3$, and get

$$\texttt{InsertionSortSteps}(3) = (3\cdot(3+1)) + \texttt{InsertionSortSteps}(3-1) = 12+10 = \boxed{22},$$

by using our previously-determined value for $\texttt{InsertionSortSteps}(2)$. Again, still finite!

In general, if this process can sort a list of size $n-1$ in finitely many steps, then it only takes us $3n+3$ more steps to sort a list of size n. In particular, there is no point at which our algorithm jumps to needing "infinitely many" steps to sort a list!

- **The algorithm produces the desired output**: in this case, it produces a list ordered from least to greatest.

  This happens! To see why, notice that in step 4, we always make the first element of the list we're currently sorting the smallest element in our list. Therefore, on our first application of step 4, we have ensured that l_1 is the smallest element of our list.

  On the second application of step 4, we were sorting the list starting from its second element: doing so ensures that l_2 is smaller than all of the remaining elements.

  On the third application of step 4, we had set the first two elements aside and were sorting the list starting from its third element: doing so ensures that l_3 is smaller than all of the remaining elements.

  . . . in general, on the k-th application of step 4, we had set the first $k-1$ elements aside and were sorting the list starting from its k-th element! Again, doing so ensures that l_k is smaller than all of the remaining elements.

So, in total, what does this mean? Well: by the above, we know that $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$, as by definition each element is smaller than all of the ones that come afterwards. In other words, this list is sorted!

$\square$

4.3 Recursion, Composition, and Algorithms

Algorithm 4.5 had an interesting element to its structure: in its fifth step, it contained a reference to itself! This sort of thing might feel like circular reasoning: how can you define an object in terms of itself?

However, if you think about it for a bit, this sort of thing is entirely natural: many tasks and processes in life consist of "self-referential" that are defined by self-reference! We call such definitions **recursive** definitions, and give a few examples here:

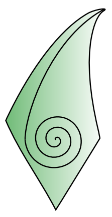
If you know some linear algebra, the explicit formulas we are using here are the following: for each point (x, y) in S, draw the four points

- $\begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6 \end{bmatrix}$,
- $\begin{bmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6 \end{bmatrix}$
- $\begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0.44 \end{bmatrix}$
- $\begin{bmatrix} 0 & 0 \\ 0 & 0.16 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

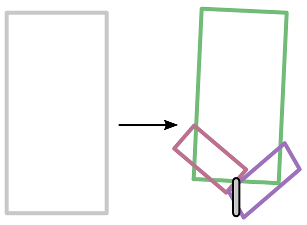
This is the precise math-y way of describing the operations we're doing to these rectangles!

Example 4.11. Fractals, and the many plants and living things that have fractal-like patterns built into theirselves, are examples of recursively-defined objects! For example, consider the following recursive process:

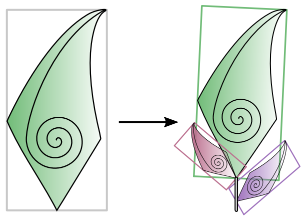
1. Start by drawing any shape S_0. For example, here's a very stylized drawing of a cluster of fern spores:



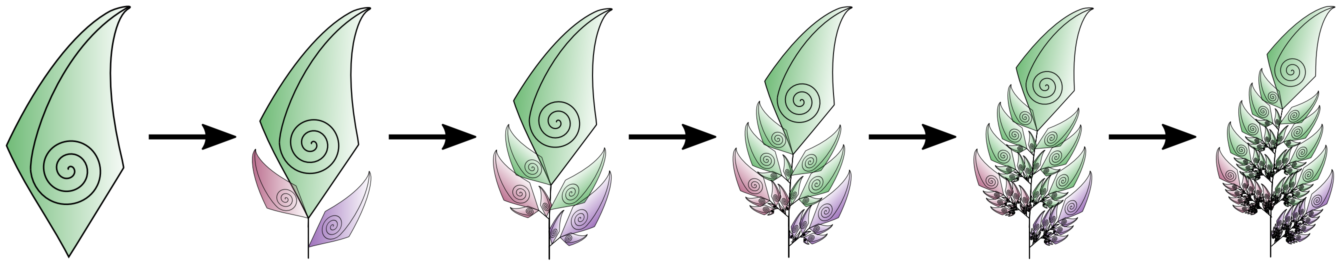
2. Now, given any shape S, we define $T(S)$ as the shape made by making four copies of S and manipulating them as follows: if S is a shape contained within the gray rectangle at left, we make four copies of S appropriately scaled/stretched/etc to match the four rectangles at right. (It can be hard to see the black rectangle, because it's so squished: it's the stem-looking bit at the bottom-middle.)



For example, $T(S_0)$, i.e. T applied to our fern spore shape, is the following:



3. Using our "seed" shape S_0 and our function T, we then **recursively** define S_n as $T(S_{n-1})$ for every positive integer n. That is: $S_1 = T(S_0), S_2 = T(S_1)$, etc. This is a recursive definition because we're defining our shapes in terms of previous shapes! Using the language of function composition, we can express this all at once by writing $S_n = \underbrace{T \circ T \circ \ldots \circ T}_{n \text{ times}}(S_0)$.
4. Now, notice what these shapes look like as we draw several of them in a row:



Our seed grows into a fern!

This is not a biology class, and there are many open questions about precisely how plants use DNA to grow from a seed. However, the idea that many plants can be formed by taking a simple instruction (given one copy of a thing, split it into appropriately stretched/placed copies of that thing) and repeatedly applying it is one that should seem reasonable, given the number of places you see it in the world!

Example 4.12. On the less beautiful but more practical side, recursion is baked into many fundamental mathematical operations! For example, think about how you'd calculate $11 \cdot 13$. By definition, because multiplication is just repeated addition, you could calculate this as follows:

$$11 \cdot 13 = \underbrace{13 + 13 + 13 + 13 + 13 + 13 + 13 + 13 + 13 + 13 + 13}_{11 \text{ times}} = 143.$$

Now, however, suppose that someone asked you to calculate $12 \cdot 13$. While you could use the process above to find this product, you could also shortcut this process by noting that

$$12 \cdot 13 = (1 + 11) \cdot 13 = 13 + 11 \cdot 13 = 13 + 143 = 156.$$

This sort of trick is essentially recursion! That is: if you want, you could define the product $n \cdot k$ recursively for any nonnegative integer n by the following two-step process:

- $0 \cdot k = 0$, for every k.
- For any $n \geq 1$, $n \cdot k = k + (n-1) \cdot k$.

The second bullet point is a recursive definition, because we defined multiplication in terms of itself! In other words, when we say that $12 \cdot 13 = 13 + 11 \cdot 13$, we're really thinking of multiplication *recursively*: we're not trying to calculate the whole thing out all at once, but instead are trying to just relate our problem to a previously-worked example.

Exponentiation does a similar thing! Because exponentiation is just repeated multiplication, we know that

$$2^{10} = \underbrace{2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2}_{10 \text{ times}} = 1024.$$

Similarly, though, we can define exponentiation recursively by saying that for any nonnegative integer n and nonzero number a, that

- $a^0 = 1$, and
- For any $n \geq 1$, $a^n = a \cdot a^{n-1}$.

In other words, with this idea, we can just say that

$$2^{11} = 2 \cdot 2^{10} = 2 \cdot 1024 = 2048,$$

instead of calculating the entire thing from scratch!

These ideas can be useful if you're working with code where you're calculating a bunch of fairly similar products/exponents/etc. With recursion, you can store some precalculated values and just do a few extra steps of working rather than doing the whole thing out by scratch each time. Efficiency!

In classes like Compsci 220 / 320, you'll study the idea of efficiency in depth, and come up with more sophisticated ideas than this one! In most practical real-life situations there are better ways to implement multiplication and exponentiation than this recursive idea; however, it can be useful in some places, and the general principle of "storing commonly-calculated values and extrapolating other values from those recursively" is one that *does* come up in lots of places!

Recursion also comes up in essentially every dynamical system that models the world around us! For instance, consider the following population modelling problem:

Example 4.13. Suppose that we have a petri dish in which we're growing a population of amoebae, each of which can be in two possible states (**small** and **large**).

Amoebas grow as follows: if an amoeba is small at some time t, then at time $t + 1$ it becomes large, by eating food around it. If an amoeba is large at some time t, then at time $t + 1$ it splits into one large amoeba and one small amoeba.

Suppose our petri dish starts out with one small amoeba at time $t = 1$. How many amoebae in total will be in this dish at time $t = n$, for any natural number n?

Answer. To help find an answer, let's make a chart of our amoeba populations over the first six time steps:

	1	2	3	4	5	6
Large	0	1	1	2	3	5
Small	1	0	1	1	2	3
Total	1	1	2	3	5	8

This chart lets us make the following observations:

1. The number of large amoebae at time n is precisely the total number of amoebae at time $n - 1$. This is because every amoeba at time $n-1$ either grows into a large amoeba, or already was a large amoeba!
2. The number of small amoebae at time n is the number of large amoebae at time $n - 1$. This is because the only source of small amoebae are the large amoebae from the earlier step when they split!
3. By combining 1 and 2 together, we can observe that the number of small amoebae at time n is the total number of amoebae at time $n - 2$!
4. Consequently, because we can count the total number of amoebae by adding the large amoebae to the small amoebae, we can conclude that **the total number of amoebae at time** n is the total number of amoebae at time $n-1$, plus the total number of amoebae at time $n - 2$. In symbols,

$$A(n) = A(n - 1) + A(n - 2).$$

We call relations like the one above **recurrence relations**, and will study them in greater depth when we get to induction as a proof method.

For now, though, notice that they are remarkably useful objects! By generalizing the model above (i.e. subtracting a_{n-3} to allow for old age killing off amoebas, or having a $-ca_n^2$ term to denote that as the population grows too large, predation or starvation will cause the population to die off, etc.) one can basically model an arbitrarily-complicated population over the long term.

These are particularly cool because they actually accurately model predator/prey relations in real life! See Isle Royale, amongst other examples.

For more details on these sorts of population-modelling processes, see the logistic map, the Lotka-Volterra equations, and most of the applied mathematics major here at Auckland!

It bears noting that the amoeba recurrence relation is not the first recurrence relation you've seen! If you go back to Algorithm 4.5, we proved that

$$\texttt{InsertionSortSteps}(n) = 3(n+1) + \texttt{InsertionSortSteps}(n-1).$$

This is another recurrence relation: it describes the maximum number of operations needed to sort a list of size n in terms of the maximum number of operations needed to sort a list of size $n-1$.

While recurrence relations are nice, they can be a little annoying to work with directly. For example, suppose that someone asked us what `InsertionSortSteps`(12) is. Because we don't have a non-recursive formula, the only thing we could do here is just keep recursively applying our formula, to get

$$\begin{aligned}
\texttt{InsertionSortSteps}(12) &= 3(12+1) + \texttt{InsertionSortSteps}(11)\\
&= 3(12+1) + 3(11+1) + \texttt{InsertionSortSteps}(10)\\
&= 3(12+1) + 3(11+1) + 3(10+1) + \texttt{InsertionSortSteps}(9)\\
&= \ldots\\
&= 3(12+1) + 3(11+1) + 3(10+1) + \ldots + 3(3+1) + 3(2+1) + \texttt{InsertionSortSteps}(1)\\
&= 3(12+1) + 3(11+1) + 3(10+1) + \ldots + 3(3+1) + 3(2+1) + 1 = \boxed{265}.
\end{aligned}$$

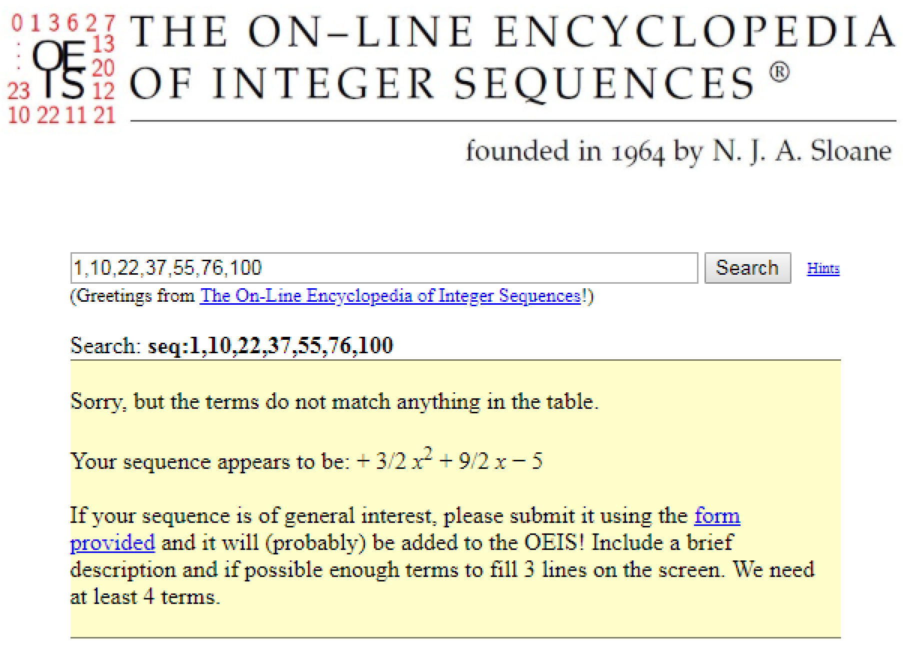
This is ...kinda tedious. It would be nice if we had a direct formula for this: something like $\texttt{InsertionSortSteps}(n) = 2^n$ or $n^2 - 4n + 17$ that we could just plug n into and get an answer.

To get these skills, either take Maths 120 and study linear algebra / eigenvalues, or take Compsci 220/320, or take Maths 326 and learn about generating functions!

At this point in time, we don't have enough mathematics to directly find such a "closed" form. However, we can still sometimes find an answer by just guessing. To be a bit more specific: if we use our definition, we can calculate the following values for `InsertionSortSteps`(n):

n	1	2	3	4	5	6	7
`InsertionSortSteps`(n)	1	10	22	37	55	76	100

If you plug this into the Online Encyclopedia of Integer Sequences, it will give you the following guess by comparing it to all of the sequences it knows:


THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES ®

founded in 1964 by N. J. A. Sloane

1,10,22,37,55,76,100 Search Hints

(Greetings from The On-Line Encyclopedia of Integer Sequences!)

Search: **seq:1,10,22,37,55,76,100**

Sorry, but the terms do not match anything in the table.

Your sequence appears to be: $+ 3/2\, x^2 + 9/2\, x - 5$

If your sequence is of general interest, please submit it using the form provided and it will (probably) be added to the OEIS! Include a brief description and if possible enough terms to fill 3 lines on the screen. We need at least 4 terms.

This turns out to be correct!

Claim 4.2. *For every positive integer n, we have* $\texttt{InsertionSortSteps}(n) = \dfrac{3n^2+9n-10}{2}$.

We don't have the techniques to prove this just yet. If you would like to see a proof, though, skip ahead to the induction section of our proofs chapter! We'll tackle this problem there (along with another recurrence relation), in Section 7.8.

4.4 Runtime and Algorithms

In the above few sections, we studied $\texttt{InsertionSortSteps}(n)$ and analyzed the maximum number of operations it needs to sort a list of length n. In general, this sort of **run-time analysis** of an algorithm — i.e. the number of elementary operations needed for that algorithm to run — is a very useful thing to be able to do!

To give a brief example of *why* this is useful, consider another, less well-known algorithm that we can use to sort a list:

Algorithm 4.6. *1 The following algorithm,* $\texttt{BogoSort}(L)$*, takes in a list $L = (l_1, l_2, \ldots l_n)$ of n numbers and orders it from least to greatest. It does this by using the following algorithm:*

1. *One by one, starting with the first entry in our list and working our way through our list, compare the values stored in consecutive elements in our list.*

   If these elements never decrease — i.e. if when we perform these comparisons, we see that $l_1 \le l_2 \le \ldots \le l_n$ — then our list is already sorted! Stop.

2. *Otherwise, our list is not already sorted. In this case, randomly shuffle the elements of our list around, and loop back to step 1.*

Here's a sample run of this algorithm on the list $(1, 7, 1, 0)$, where I've used random.org to shuffle our list when needed by step 3:

list	iteration count	step	sorted
(1,7,1,0)	1	1	no
(1,1,0,7)		2	
(1,1,0,7)	2	1	no
(7,0,1,1)		2	
(7,0,1,1)	3	1	no
(1,0,1,7)		2	
(1,0,1,7)	4	1	no
(7,1,1,0)		2	
(7,1,1,0)	5	1	no
(7,0,1,1)		2	
(7,0,1,1)	6	1	no
(1,1,0,7)		2	
(1,1,0,7)	7	1	no
(1,1,7,0)		2	
(1,1,7,0)	8	1	no
(0,1,7,1)		2	
(0,1,7,1)	9	1	no
(7,1,0,1)		2	
(7,1,0,1)	10	1	no
(1,7,0,1)		2	
(1,7,0,1)	11	1	no
(1,7,1,0)		2	
(1,7,1,0)	12	1	no
(1,7,1,0)		2	
(1,7,1,0)	13	1	no
(0,1,7,1)		2	
(0,1,7,1)	14	1	no
(0,1,1,7)		2	
(0,1,1,7)	15	1	yes!

This . . . is not great. If you used `BogoSort` to sort a deck of cards, your process would look like the following:

- One-by-one, go through your deck of cards and see if they're ordered.
- If during this process you spot any cards that are out of order, throw the whole deck in the air, collect the cards together, and start again.

By studying the running time of this algorithm, we can make "not great" into something rigorous:

Claim 4.3. *The worst-case running time for `BogoSort` to sort any list containing more than one element is ∞.*

Proof. If L is a list containing n different elements, there are $n!$ many different ways to order L's elements (this is ordered choice without repetition, where we think of ordering our list as "choosing" elements one-by-one to put in order.) If all of these elements are different, then there is exactly one way for us to put these elements in order.

Therefore, on each iteration `BogoSort` has a $\frac{1}{n!}$ chance of successfully sorting our list, and therefore has a $\frac{n!-1}{n!}$ chance of **failing** to sort our list. For any $n > 1$, $n!-1$ is nonzero, and so the chance that our algorithm fails on any given iteration is nonzero.

Therefore, in the worst-case scenario, it is possible for our algorithm to just fail on each iteration, and thereby this algorithm could have infinite run-time. □

In terms of running time, then, we've shown that `BogoSort` has a strictly worse runtime than `InsertionSort`, as $\infty > \dfrac{3n^2+9n^{2^2}-10}{2}$. Success!

This sort of comparison was particularly easy to perform, as one of the two things we were comparing was ∞. However, this comparison process can get trickier if we examine more interesting algorithms. Let's consider a third sorting algorithm:

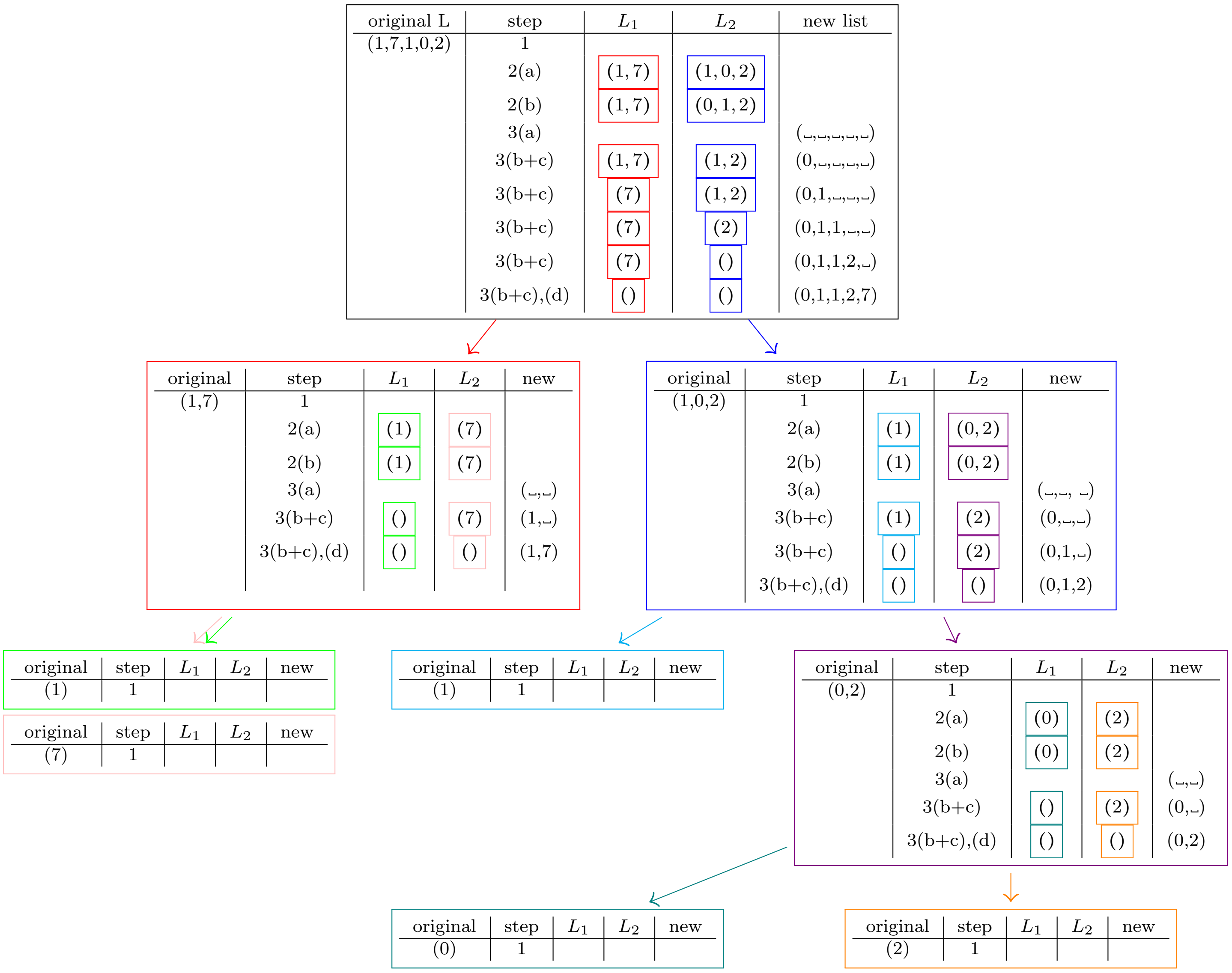
Algorithm 4.7. *The following algorithm, `MergeSort`(L), takes in a list $L = (l_1, l_2, \dots l_n)$ of n numbers and orders it from least to greatest. It does this by using the following algorithm:*

1. *If L contains at most one number, L is trivially sorted! In this situation, stop.*
2. *Otherwise, L contains at least two numbers. In this case,*
   - *(a) Split L in half into two lists L_1, L_2.*
   - *(b) Apply `MergeSort` to each of L_1, L_2 to sort them.*
3. *Now, we "merge" these two sorted lists:*
   - *(a) Create a big list with n entries in it, all of which are initially blank.*
   - *(b) Compare the first element in L_1 to the first element in L_2. If L_1, L_2 are both sorted, these first elements are the smallest elements in L_1, L_2.*

(c) Therefore, the smaller of those two first elements is the smallest element in our entire list. Take it, remove it from the list it came from, and put it in the first blank location in our big list.

(d) Repeat (b)+(c) until our big list is full!

As before, to better understand this algorithm, let's run it on an example list like $(1,7,1,0,2)$:



original L	step	L_1	L_2	new list
(1,7,1,0,2)	1			
	2(a)	$(1,7)$	$(1,0,2)$	
	2(b)	$(1,7)$	$(0,1,2)$	
	3(a)			(␣,␣,␣,␣,␣)
	3(b+c)	$(1,7)$	$(1,2)$	(0,␣,␣,␣,␣)
	3(b+c)	(7)	$(1,2)$	(0,1,␣,␣,␣)
	3(b+c)	(7)	(2)	(0,1,1,␣,␣)
	3(b+c)	(7)	()	(0,1,1,2,␣)
	3(b+c),(d)	()	()	(0,1,1,2,7)

original	step	L_1	L_2	new
(1,7)	1			
	2(a)	(1)	(7)	
	2(b)	(1)	(7)	
	3(a)			(␣,␣)
	3(b+c)	()	(7)	(1,␣)
	3(b+c),(d)	()	()	(1,7)

original	step	L_1	L_2	new
(1,0,2)	1			
	2(a)	(1)	$(0,2)$	
	2(b)	(1)	$(0,2)$	
	3(a)			(␣,␣, ␣)
	3(b+c)	(1)	(2)	(0,␣,␣)
	3(b+c)	()	(2)	(0,1,␣)
	3(b+c),(d)	()	()	(0,1,2)

original	step	L_1	L_2	new
(1)	1			

original	step	L_1	L_2	new
(7)	1			

original	step	L_1	L_2	new
(1)	1			

original	step	L_1	L_2	new
(0,2)	1			
	2(a)	(0)	(2)	
	2(b)	(0)	(2)	
	3(a)			(␣,␣)
	3(b+c)	()	(2)	(0,␣)
	3(b+c),(d)	()	()	(0,2)

original	step	L_1	L_2	new
(0)	1			

original	step	L_1	L_2	new
(2)	1			

Here, we use the colored boxes to help us visualize the recursive applications of `MergeSort` to smaller and smaller lists.

As before, it is worth taking a moment to explain *why* this algorithm works:

Claim 4.4. *`MergeSort` (i.e. algorithm 4.7) works.*

Proof. We proceed in the same way as when we studied `InsertionSort`:

- **Does the algorithm have any bugs?** Nope! The only things we do in our algorithm are compare elements, split our lists, and copy elements over. Those are all well-defined things that can be done without dividing by zero or other sorts of disallowed operations!
- **Does the algorithm run forever?** Nope!

To see why, let's make a recurrence relation like we did before. To be precise:

Claim 4.5. *If we let* $\mathtt{MergeSortSteps}(n)$ *denote the maximum number of steps needed by* `MergeSort` *to sort a list of n elements, then* $\mathtt{MergeSortSteps}(n) = 1 + 4n + 2\mathtt{MergeSortSteps}(n/2)$ *if n is even.*

Proof. By definition, `MergeSort` performs the following operations:

- In step 1, it performs one operation (it looks up the size of the list.)
- In step 2, it splits the list in half. We can do this with at most n write operations by just making two blank lists of size $n/2$ and copying elements over one-by-one to these new lists. It then runs `MergeSort` on each half. Doing this gives us an extra $2 \cdot \mathtt{MergeSortSteps}(n/2)$ steps.
- In step 3, we repeatedly compare the first element in L_1 to the first element in L_2, remove the smaller of the two elements from that list, and put it into our big list. This is one comparison and two write operations, which we did as many times as we had elements in our original list; so we have at most $3 \cdot n$ operations here in total.

If n is odd, this gets a bit more annoying and we'd have `MergeSortSteps`("$n/2$ rounded up") plus `MergeSortSteps`("$n/2$ rounded down") here instead. This doesn't materially change things, but it is annoying enough to warrant just looking at the even cases.

In total, then, we have

$$\begin{aligned}\mathtt{MergeSortSteps}(n) &= 1 + n + 2 \cdot \mathtt{MergeSortSteps}(n/2) + 3 \cdot n\\ &= \boxed{1 + 4n + 2 \cdot \mathtt{MergeSortSteps}(n/2)}.\end{aligned}$$

□

If we just round any odd-length list up to an even-length list by adding a blank cell, this gives us a recurrence relation that lets us reduce the task of calculating $\mathtt{MergeSortSteps}(n)$ for any n to the task of calculating smaller values of $\mathtt{MergeSortSteps}(n)$. Therefore this process cannot run forever, by the same reasoning as with $\mathtt{InsertionSortSteps}(n)$!

- **Does the algorithm sort our list?** Yes!

  To see why, make the following observations:

  - Our algorithm trivially succeeds at sorting any list with one element.
  - Our algorithm also succeeds at sorting any list with 2 elements. To see why, note that by definition, it takes those two elements and splits them into two one-element lists. From there, it puts the smaller element from those two lists into the first position, and puts the larger in the second position, according to our rules. That's a sorted list!
  - Now, consider any list on 3 or 4 elements. By definition, our algorithm will do the following:
    - ∗ It takes that list, and splits it into two lists of size 1 or 2.
    - ∗ It sorts those lists (and succeeds, by our statements above!)
    - ∗ It then repeatedly takes the smaller of the first elements in either of those two lists, removes it from that list, and puts it into our larger list. Because those small lists are sorted, each time we do this we're removing the smallest unsorted element (as the first element in a sorted list is its smallest element.) Therefore, this process puts elements into our largest list in order by size, and thus generates a sorted list.

So our process works on lists of 3 or 4 elements!

- The same logic will tell us that our process works for lists on 5-8 elements: because any list on 5-8 elements will split into two lists of size at most 4, and our process works on lists of size at most 4, it will continue to succeed!
- In general, our process will always work! This is because our process works by splitting n in half, and thus it *reduces* the task of sorting a list of size n into the task of sorting two lists of size $\approx n/2$. Repeatedly performing this reduction eventually reduces our task to just sorting a bunch of small lists, which we've shown here that we can do.

□

As well, by using the same sort of "guess-a-pattern" idea from before, we can refine this to a closed-form solution!

Note that if we use our definition, we can calculate the following values for $\texttt{MergeSortSteps}(2^k)$. By definition, $\texttt{MergeSortSteps}(1) = 1$, as this algorithm stops when given any list of length 1. Therefore, by repeatedly using Claim 4.5, we can get the following:

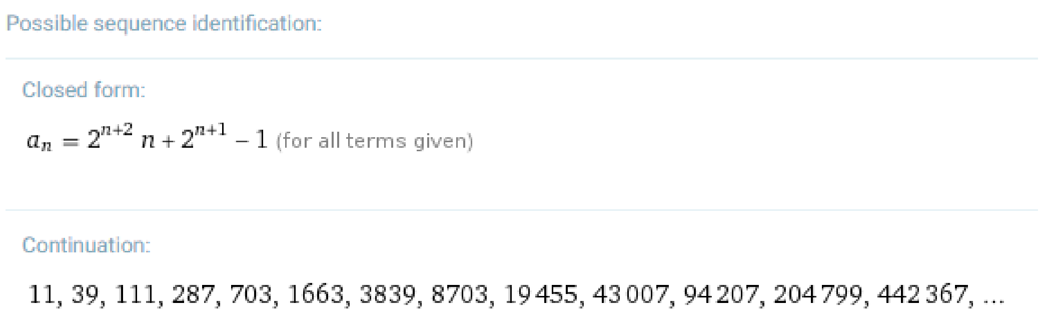
$$\texttt{MergeSortSteps}(2^1) = 1 + 4\cdot 2 + 2\cdot\texttt{MergeSortSteps}(2/2) = 1 + 8 + 2\cdot\texttt{MergeSortSteps}(1) = 11,$$
$$\texttt{MergeSortSteps}(2^2) = 1 + 4\cdot 4 + 2\cdot\texttt{MergeSortSteps}(4/2) = 1 + 16 + 2\cdot\texttt{MergeSortSteps}(2) = 39,$$
$$\texttt{MergeSortSteps}(2^3) = 1 + 4\cdot 8 + 2\cdot\texttt{MergeSortSteps}(8/2) = 1 + 32 + 2\cdot\texttt{MergeSortSteps}(4) = 111,$$
$$\texttt{MergeSortSteps}(2^4) = 1 + 4\cdot 16 + 2\cdot\texttt{MergeSortSteps}(16/2) = 1 + 64 + 2\cdot\texttt{MergeSortSteps}(8) = 287,$$
$$\texttt{MergeSortSteps}(2^5) = 1 + 4\cdot 32 + 2\cdot\texttt{MergeSortSteps}(32/2) = 1 + 128 + 2\cdot\texttt{MergeSortSteps}(16) = 703,$$
$$\texttt{MergeSortSteps}(2^6) = 1 + 4\cdot 64 + 2\cdot\texttt{MergeSortSteps}(64/2) = 1 + 256 + 2\cdot\texttt{MergeSortSteps}(32) = 1663,$$
$$\texttt{MergeSortSteps}(2^7) = 1 + 4\cdot 128 + 2\cdot\texttt{MergeSortSteps}(128/2) = 1 + 512 + 2\cdot\texttt{MergeSortSteps}(64) = 3839,$$

which in table form is the following:

k	1	2	3	4	5	6	7
$\texttt{MergeSortSteps}(2^k)$	11	39	111	287	703	1663	3839

(Note that we're using 2^k as the length of our lists. This is because our result only works for even numbers, so we want something that stays even when we keep dividing it by 2.)

Spotting the pattern here is a pain without more advanced mathematics; even the Online Encyclopedia of Integer Sequences doesn't recognize it. WolframAlpha, however, does!



Possible sequence identification:

Closed form:

$a_n = 2^{n+2}\, n + 2^{n+1} - 1$ (for all terms given)

Continuation:

11, 39, 111, 287, 703, 1663, 3839, 8703, 19 455, 43 007, 94 207, 204 799, 442 367, ...

Again, this is a claim whose proof will have to wait for Section 7.8. For now, though, let's take the following as given:

Claim 4.6. *$\texttt{MergeSortSteps}(2^k) = k\cdot 2^{k+2} - 2^{k+1} - 1$, for every natural number k.*

We can easily extend this to a list whose length is not a power of two by just "rounding its length up to the nearest power of 2" by adding in some blank cells: i.e. if we had a list of length 28, we'd add in 4 blank cells to get a list of length 32.

If we do this, then Claim 4.6 becomes the following:

Observation 4.10. *$\texttt{MergeSortSteps}(n) \le k \cdot 2^{k+2} - 2^{k+1} - 1$, where 2^k is n rounded up to the nearest power of 2.*

Recall that $\lceil x \rceil$ is "x rounded up."

To simplify this a bit to just write things in terms of n, notice that if 2^k is n rounded up to the nearest power of 2, then $k = \lceil \log_2(n) \rceil$. Plug this into Observation 4.10, and you'll get the following:

Two useful tricks we're using in this calculation:

- $x \le \lceil x \rceil$. That is: rounding a number up only makes it larger.
- $\lceil x \rceil < x + 1$. That is: rounding up never increases a number by more than 1.

$$\begin{aligned}
\texttt{MergeSortSteps}(n) &\le \lceil \log_2(n) \rceil \cdot 2^{\lceil \log_2(n) \rceil + 2} - 2^{\lceil \log_2(n) \rceil + 1} - 1 \\
&\le (\log_2(n) + 1) 2^{(\log_2(n)+1)+2} - 2^{\log_2(n)+1} - 1 \\
&= (\log_2(n) + 1) 2^{\log_2(n)} \cdot 2^3 - 2^{\log_2(n)} \cdot 2 - 1 \\
&= 8n \log_2(n) + 8n - 2n - 1 \\
&= \boxed{8n \log_2(n) + 6n - 1.}
\end{aligned}$$

Nice! This is worth making into its own observation:

Observation 4.11. *$\texttt{MergeSortSteps}(n) \le 8n \log_2(n) + 6n - 1$, for every positive integer $n > 1$.*

4.5 Comparing Runtimes: Limits

Now, we have an interesting problem on our hands: between `MergeSort` and `InsertionSort`, which of these two algorithms is the most efficient? That is: in terms of functions, which of the following is better?

$$\boxed{8n \log_2(n) + 6n - 1} \quad \text{vs.} \quad \boxed{\frac{3n^2 + 9n - 10}{2}}$$

To start, we can make a table to compare values:

n	1	2	3	4	5	6	7
$\frac{3n^2 + 9n - 10}{2}$	1	10	22	37	55	76	100
$8n \log_2(n) + 4n - 1$	5	27	54.0	87	121.9	159.1	198.2

It looks like the `MergeSort` algorithm needs more steps than `InsertionSort` so far. However, this table only calculated the first few values of n; in real life, however, we often find ourselves sorting *huge* lists! So: in the long run, which should we prefer?

To answer this, we need the idea of a **limit**:

If you would like a more rigorous definition here than "gets close," look into classes like Maths 130 and Maths 250!

Definition 4.4. *Given any function f that is defined on the natural numbers $\mathbb{N}$, we say that $\lim_{n \to \infty} f(n) = L$ if " as n goes to infinity, $f(n)$ gets closer and closer to L." In particular, we say that $\lim_{n \to \infty} f(n) = +\infty$ if "as n goes to infinity, $f(n)$ also grows without bound."*

This is a tricky concept! To understand it, let's look at a handful of examples:

Problem 4.1. What is $\lim_{n\to\infty} \dfrac{1}{2 - \frac{1}{n}}$?

Answer. First, notice that as n goes to infinity, $\frac{1}{n}$ goes to 0, because its denominator is going to infinity while the numerator stays fixed.

Therefore, $2 - \frac{1}{n}$ goes to 2, and so $\lim_{n\to\infty} \dfrac{1}{2 - \frac{1}{n}} = \dfrac{1}{2}$.

Problem 4.2. What is $\lim_{n\to\infty} \log_2(n)$?

Answer. $\lim_{n\to\infty} \log_2(n) = \infty$. This is because for any positive integer k, we can make $\log_2(n) \geq k$ by setting n to be any value $\geq 2^k$.

In other words, as n grows, it eventually gets larger than 2^k for any fixed k, and thus $\log_2(n)$ itself grows without bound.

Problem 4.3. What is $\lim_{n\to\infty} \dfrac{1}{\log_2\left(\frac{1}{n}\right)}$?

Answer. To find this limit, we simply break our function down into smaller pieces:

- First, notice that as n takes on increasingly large positive values, $\frac{1}{n}$ goes to 0 and stays positive.
- Therefore, $\log_2\left(\frac{1}{n}\right)$ goes to negative infinity, as $\log_2$(tiny positive numbers) yields increasingly huge negative numbers.
- Therefore, $\frac{1}{\log_2\left(\frac{1}{n}\right)}$ goes to 0, as $\frac{1}{\text{huge negative numbers}}$ yields tiny negative numbers.

In total, then, $\lim_{n\to\infty} \dfrac{1}{\log_2\left(\frac{1}{n}\right)} = 0$.

Problem 4.4. What is $\lim_{n\to\infty} \dfrac{n^2 + 3n - 4}{n^3 + 2}$?

Answer. It is tempting to just "plug in infinity" into the fraction above, and say that

$$\text{“because } \tfrac{\infty^2 + 3\infty - 4}{\infty^3 + 2} = \tfrac{\infty}{\infty} = 1, \text{ our limit is 1”}$$

However, you can't do manipulations like this with infinity! For example, because $\frac{1}{n} = \frac{n}{n^2}$, we have

$$\lim_{n\to\infty} \frac{n}{n^2} = \lim_{n\to\infty} \frac{1}{n} = 0,$$

even though the method above would say that

$$\text{“because } \tfrac{\infty}{\infty^2} = \tfrac{\infty}{\infty} = 1, \text{ our limit is 1.”}$$

The issue here is that there are different growth rates at which various expressions approach infinity: i.e. in our example above, n^2 approaches infinity considerably faster than n, and so the ratio $\frac{n}{n^2}$ approaches 0 even though the numerator and denominator individually approach infinity.

Instead, if we ever have both the numerator and denominator approaching $+\infty$, we need to first **simplify** our fraction to proceed further! In this problem, notice that if we divide both the numerator and the denominator by n^3, the highest power present in either the numerator or denominator, we get the following:

$$\frac{n^2 + 3n - 4}{n^3 + 2} \cdot \frac{1/n^3}{1/n^3} = \frac{\frac{1}{n} + \frac{3}{n^2} - \frac{4}{n^3}}{1 + \frac{2}{n^3}}.$$

As noted above, each of $\frac{1}{n}, \frac{3}{n^2}, -\frac{4}{n^3}, \frac{2}{n^3}$ go to 0 as n goes to infinity, because their denominators are going off to infinity while the numerator is staying fixed.

Therefore, we have

$$\lim_{n\to\infty} \frac{n^2+3n-4}{n^3+2} = \lim_{n\to\infty} \frac{\frac{1}{n}+\frac{3}{n^2}-\frac{4}{n^3}}{1+\frac{2}{n^3}} = \lim_{n\to\infty} \frac{0+0+0}{1+0} = \boxed{0}.$$

With the idea of limits in mind, we can now talk about how to compare functions:

Definition 4.5. *Let $f(n)$ and $g(n)$ be two functions which depend on n. We say that the function $f(n)$* **grows faster** *than the function $g(n)$ if $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = +\infty$.*

Intuitively, this definition is saying that for huge values of n, the ratio of $f(n)$ to $g(n)$ goes to infinity: that is, $f(n)$ is eventually as many times larger than $g(n)$ as we could want.

We work an example of this idea here, to see how it works in practice:

Example 4.14. We claim that the function $f(n) = n^2+2n+1$ grows faster than $g(n) = 5n+5$. To see this, by our definition above, we want to look at the limit $\lim_{n\to\infty} \frac{n^2+2n+1}{5n+5}$.

Notice that we can factor the numerator into $(n+1)^2$. Plugging this into our fraction leaves us with $\frac{(n+1)^2}{5(n+1)}$, which we can simplify (as in Problem 4.4,) to $\frac{n+1}{5}$.

As n goes to infinity, $\frac{n+1}{5}$ goes to infinity.

This grows without bound: as n goes to infinity, so does $\frac{1}{5}n+\frac{1}{5}$! Therefore we've shown that n^2+2n+1 grows faster than $5n+5$.

To deal with a comparison like the one we're trying to do between `MergeSort` and `InsertionSort`, however, we need some more tricks!

4.6 Limit Techniques and Heuristics

Without calculus, our techniques for limits are a little hamstrung. As those of you who have seen NCEA L3 calculus, things like L'Hôpital's rule are extremely useful for quickly evaluating limits!

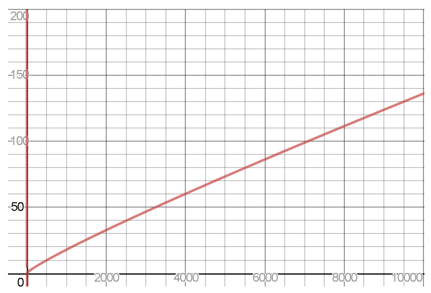
With that said, though, we do have a handful of useful techniques and heuristics that we can use to get by. Here's an easy (if not very rigorous) technique:

Observation 4.12. ***Plugging In Values.*** *Probably the simplest thing you can do, when given a limit, is just physically plug in numbers and figure out where the function is going.*

For instance, suppose that we wanted to compare the runtime of our two functions `MergeSortSteps` and `InsertionSortSteps`. By definition, this means that we want to find the limit

$$\lim_{n\to\infty} \frac{\frac{3n^2+9n-10}{2}}{8n\log_2(n)+4n-1}$$

To do this, we could just plug in various values of n and see what happens!



x	$f(x)$
1	0.33
10	0.63
100	2.70
1000	17.97
10000	136.03

It certainly looks like $f(x)$ is growing arbitrarily large, so we could quite reasonably believe that the number of steps required to calculate `InsertionSort` is growing faster than the number of steps needed to calculate `MergeSort` (and thus `MergeSort` is the preferable algorithm.)

However, this method has its limitations:

- One issue with the above is that it is fairly prone to human error if you're manually doing this by plugging numbers into a calculator. That is: if you have a limit like $\lim\limits_{x\to\infty} \dfrac{x^3 - 3x^2 + 3x}{x^2 - x}$, and you're trying to plug in something like $x = 1000000$ into that fraction, it's going to be really easy to forget a zero in one of those x expressions.
- Another is that it can be pretty hard to tell whether or not you're actually plugging in enough values to figure out the pattern! For example, when we made our table to compare the runtimes of these two functions by listing values from 1 through 7, we thought that `MergeSortSteps` was growing faster. This larger table seems to be telling the opposite story: but maybe the situation will reverse itself again if we zoom out further!

  To give a second example, suppose that someone asked you to calculate

  $$\lim_{n\to\infty} \log_2(\log_2(\log_2(\log_2(\log_2(n)))))$$

  If you plugged in $n = 100, 1000, 10000,$, you'd get $\approx -0.9, -0.34, -0.11$. This looks like it's slowly increasing to 0, so you'd be tempted to guess that

  $$\lim_{n\to\infty} \log_2(\log_2(\log_2(\log_2(\log_2(n))))) = 0.$$

  This is *very false*! As we saw before, as n goes to infinity, $\log_2(n)$ goes to infinity. Therefore, $\log_2(\log_2(n))$ also goes to infinity, and in general any composition of logs will eventually go off to infinity as well: i.e. $+\infty$ is the correct limit here. So plugging things in can lead us to make mistakes!

A second technique, that we used in several of our worked problems earlier, is the following:

Observation 4.13. ***Simplifying Fractions.*** *In the special case where your limit has the form* $\lim\limits_{x\to\infty} \dfrac{f(x)}{g(x)}$, *a useful technique you can try is* ***simpliflication!*** *Basically, take your fraction* $\dfrac{f(x)}{g(x)}$, *and try to simplify it by factoring the top and bottom and canceling terms.*

This often gives you a new function where you no longer have the top and bottom going to zero, which is often much easier to work with.

Example 4.15. If we had the limit $\lim\limits_{x \to +\infty} \frac{x^2 - x}{x}$, we could factor an x out of the top and bottom to get $\lim\limits_{x \to \infty} x - 1$. This is $+\infty$.

For a second example, if we had the limit $\lim\limits_{x \to \infty} \frac{x^2 - x}{x^2}$, we could factor an x^2 out of the top and bottom to get the simplified limit $\lim\limits_{x \to +\infty} \frac{1 - \frac{1}{x}}{1}$. The numerator goes to 1 and the denominator just is 1, so this is 1.

Sometimes, however, we don't have something that we can easily express as a fraction:

Problem 4.5. What is the limit

$$\lim_{x \to +\infty} \frac{1}{1 - 2^{1/x}}$$

To approach this limit, we need another technique:

Observation 4.14. ***Break It Down.*** *Given a limit $\lim\limits_{x \to +\infty} f(x)$, we can often figure out what's going on with it by breaking our functions down into small pieces, looking at what those individual pieces do as x goes to $+\infty$, and then slowly "zooming back out" to see what the whole function does.*

To understand this method, let's apply it to Problem 4.5.

Answer to Problem 4.5. Understanding the function in our limit all at once is hard! But, notice that for very large values of x, we know that

- $1/x$ is a very small positive number, therefore
- $2^{1/x}$ is basically $2^{\text{basically 0, positive}}$, which is slightly larger than 1, therefore
- $1 - 2^{1/x}$ is basically $1 -$ (slightly larger than 1), and so in turn is a tiny negative number, therefore
- $\frac{1}{1 - 2^{1/x}}$ is 1 over a tiny negative number, and thus is an increasingly huge negative number.

Therefore, as x goes to positive infinity $\frac{1}{1 - 2^{1/x}}$ goes to $-\infty$, and we've found our limit by breaking our function down into smaller pieces!

Observation 4.15. ***Heuristics.*** *Our last limit idea is the following: with some calculus (take Maths 130!), you can prove that*

$$\textit{Constants} << \textit{Logarithms} << \textit{Polynomials} << \textit{Exponentials} << \textit{Factorials},$$

where "<<" means "grows slower than."

Within those groups, we sort these expressions by degrees and bases: i.e.

$$n << n^2 << n^3 << n^4 << \ldots,$$

and

$$2^n << 3^n << 4^n << 5^n << \ldots,$$

and so on / so forth.

Finally, any sum of expressions grows as fast as its largest expression: i.e. a factorial plus a log plus a constant grows at a factorial rate, a n^4 plus a log plus a square root grows as fast as n^4 grows, etc.

As a brief justification for this, let's look at a table:

Runtime vs. Input	10	20	30	40	50
1	$1 \cdot 10^{-6}$ sec.	$1 \cdot 10^{-6}$ sec.	$1 \cdot 10^{-6}$ sec.	$1 \cdot 10^{-6}$ sec.	$1 \cdot 10^{-6}$ sec.
$\log_2(n)$	$3.3 \cdot 10^{-6}$ sec.	$4.3 \cdot 10^{-6}$ sec.	$4.9 \cdot 10^{-6}$ sec.	$5.3 \cdot 10^{-6}$ sec.	$5.6 \cdot 10^{-6}$ sec.
n	$1 \cdot 10^{-5}$ sec.	$2 \cdot 10^{-5}$ sec.	$3 \cdot 10^{-5}$ sec.	$4 \cdot 10^{-5}$ sec.	$5 \cdot 10^{-5}$ sec.
n^2	$1 \cdot 10^{-4}$ sec.	$4 \cdot 10^{-4}$ sec.	$9 \cdot 10^{-4}$ sec.	$1.6 \cdot 10^{-3}$ sec.	$2.5 \cdot 10^{-3}$ sec.
n^3	$1 \cdot 10^{-3}$ sec.	$8 \cdot 10^{-3}$ sec.	.027 sec.	.064 sec.	.125 sec.
n^5	.1 sec.	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.
2^n	$1 \cdot 10^{-3}$ sec.	1 sec.	17.9 min.	12.7 days.	35.7 years
$n!$	3.63 sec.	77146 years	$8.4 \cdot 10^{18}$ years	$2.6 \cdot 10^{34}$ years	$9.6 \cdot 10^{50}$ years

Above, we've plotted five algorithms with runtimes $1, \log_2(n), n, n^2, n^3, n^5, 2^n, n!$ versus input sizes for n ranging from 10 to 50, with the assumption that we can perform one step every 10^{-6} seconds.

In this table, you can make the following observations:

- The constant-runtime algorithm is great!
- The logarithmic-runtime algorithm is also pretty great!
- The polynomial-runtime algorithms all take longer than the constant or log algorithm, but are all at least reasonable.
- The exponential runtime algorithm starts off OK, but gets horrible fast...
- ...but is somehow not as bad as the factorial-runtime algorithm, which is almost immediately unusable for any value of n.

We can use this heuristic to quickly answer our question about which of `MergeSortSteps` and `InsertionSortSteps` is preferable, without having to plug in any numbers at all!

Claim 4.7. *`InsertionSortSteps` grows faster than `MergeSortSteps`.*

Proof. As noted before, by definition, we want to find the limit

$$\lim_{n\to\infty} \frac{\frac{3n^2+9n-10}{2}}{8n\log_2(n) + 6n - 1}.$$

By dividing through both sides by n, this means that we're looking at the ratio

$$\lim_{n\to\infty} \frac{1.5n + 4.5 - \frac{5}{n}}{8\log_2(n) + 6 - \frac{1}{n}}.$$

The top is a linear expression (i.e. polynomial of degree 1), as the largest-growing object in the numerator is the $1.5n$ term. The bottom, conversely, is a logarithmic expression, as the fastest-growing object in the denominator is the $8\log_2(n)$ term.

Linear expressions grow much faster than logs, so our "plug things in" step didn't lie: `InsertionSortSteps` is growing faster than `MergeSortSteps` (and thus `MergeSort` is the preferable algorithm, as in the long run it will need to perform less operations to get to the same answer.) ☐

To study one more example of this idea and finish our chapter, let's use this principle to answer our last exercise:

Answer to 4.1. In this problem, we had two multiplication algorithms and wanted to determine which is best.

Algorithm 4.1 calculated $a \cdot b$ by basically just calculating $\overbrace{b + b + \ldots + b}^{a \text{ times}}$. As such, its runtime is pretty much just some constant times a; we need a iterations of this process to calculate the answer here, and we can let our constant be the number of steps we perform in each iteration.

Conversely, Algorithm 4.2 ran by taking a and repeatedly subtracting 1 from a and dividing a by 2 until it was 0 (while doing some other stuff.)

For the purpose of this problem, we don't have to think about *why* this process works, though look at practice problem 6 if you're curious! Instead, if we simply take it on faith that this algorithm *does* work, it's easy to calculate how long it takes to complete: it will need as many iterations as it takes to reduce a to 0 by these operations.

In the worst-case scenario for the number of iterations, we only ever divide by 2; i.e. a is a power of 2. In this case, it takes k iterations if $a = 2^k$; i.e. we need $\log_2(a)$ iterations, and thus need a constant times $\log_2(a)$ many operations.

As we saw before, logarithms grow much slower than linear functions! Therefore, Algorithm 4.2 is likely the better algorithm to work with.

4.7 Practice Problems

1. (-) Your two friends, Jiawei and Francis, have written a pair of algorithms to solve a problem. On an input of size n, Jiawei's algorithm needs to perform $n^2 + 2n$ calculations to find the answer, while Francis's algorithm needs to perform $\log_2(n) + 2^n$ calculations to find the same answer.

   Jiawei says that their algorithm is faster, because their runtime is polynomial while Francis's algorithm has an exponential in it. Francis says that their algorithm is faster, because their algorithm has a logarithm in it while Jiawei's is polynomial.

   Who is right, and why?

2. (-) Let $f(x) = 10^x + 1, g(x) = x - 3, h(x) = \log_{10}(x^3 + 3)$ and $j(x) = \sqrt[3]{x}$. Calculate the following compositions:

   - $f \circ h(x)$.
   - $g \circ f(x)$.
   - $h \circ j(x)$.
   - $f \circ g(x)$.

3. We say that a function $f : A \to B$ has an **inverse** if there is some function $f^{-1} : B \to A$ such that $f \circ f^{-1}(x) = x = f^{-1} \circ f$ for all x: that is, f^{-1} "undoes" f, and vice-versa. For example, $\log_2(x)$ and 2^x are inverses, as is x^5 and $\sqrt[5]{x}$.

   - Suppose that $f(x) = 3x + 2$. Find $f(x)$'s inverse.
   - Now, suppose that $g(x) = 3$. Does this constant function have an inverse?

4. Let A be the collection of all students enrolled at the University of Auckland and B be the collection of classes currently running at the University of Auckland. Which of the following are functions?

   - The rule $f : A \to B$, that given any student outputs the classes they're enrolled in.
   - The rule $f : B \to A$, that given any class outputs the student with the top mark in that class.
   - The rule $f : A \to A$, that given any student outputs that student.
   - The rule $f : B \to B$, that given any class outputs its prerequisites.

5. You're a programmer! You've found yourself dealing with a program `puzzle(n)` that has no comments in its code, and you want to know what it does. After some experimentation, you've found that `puzzle(n)` takes in as input an integer n, and does the following:

(i) Take n and square it.
(ii) If n is 1, 2, 3 or 6, output n and stop.
(iii) Otherwise, replace n with $(n^2)\,\%\,10$, i.e. the last digit of n^2, and go to (ii).

- Is `puzzle(n)` a function, if we think of its domain and codomain as $\mathbb{Z}$?
- What is the range of `puzzle(n)`?

6. (+) Show that after each step in Algorithm 4.2, the quantity $\boxed{\mathbf{a}\cdot\mathbf{b}+prod}$ is always the same. Using this, prove that Algorithm 4.2 works!
7. (+) In our answer to 4.1, we used a sort of handwave-y "this takes about $\log_2(a)$ many operations" argument. Let's make this more rigorous!

   That is: let `clevermultsteps`(a) denote the number of steps that this algorithm needs to multiply a given number b by a.

   (a) Find a recurrence relation `clevermultsteps`(a) in terms of `clevermultsteps`$(a/2)$, that holds whenever a is even.

   (b) Use this relation to calculate `clevermultsteps`(a) for $a = 2, 4, 8, 16, 32$, and plug these values into either the OEIS or WolframAlpha. What pattern do you see?
8. Consider the following sorting algorithm, called `BubbleSort`:

   Algorithm 4.8. *The following algorithm, `BubbleSort`(L), takes in a list $L = (l_1, l_2, \ldots l_n)$ of n numbers and orders it from least to greatest. It does this by using the following algorithm:*

   (i) Compare l_1 to l_2. If $l_1 > l_2$, swap these two values; otherwise, leave them where they are.
   (ii) Now, move on, and compare the "next" pair of adjacent values l_2, l_3. Again, if these elements are out of order (i.e. $l_2 > l_3$) swap them.
   (iii) Keep doing this through the entire list!
   (iv) At the end of this process, if you made no swaps, stop: your list is in order, by definition. Stop!
   (v) Otherwise, the list might be out of order! Return to (i).

   (a) Use this process to sort the list $(6, 1, 4, 2, 3, 2)$.

   (b) (+) Let `BubbleSortSteps`(n) denote the maximum number of steps that `BubbleSort` needs to sort a list of length n. Find an expression for `BubbleSortSteps`(n), and calculate its values for $n = 1, 5, 10$ and 100.

   (c) What is the *smallest* number of steps that `BubbleSort` would need to sort a list of length n?
9. Find the following limits:

   (a) $\displaystyle\lim_{n\to\infty} \frac{2^n + 2^{n+1}}{2^n - 2^{n-1}}$

   (b) $\displaystyle\lim_{n\to\infty} \frac{n^3 + 3n^2 + 3n + 1}{n^3 - 3n^2 + 3n - 1}$

   (c) $\displaystyle\lim_{n\to\infty} 2^{\log_2(n)-6}$

   (d) $\displaystyle\lim_{n\to\infty} \frac{\sin(2^n + n) + n}{n - \log_2(n)}$

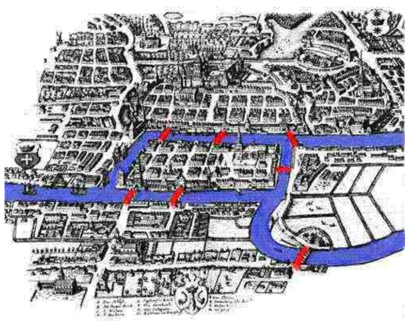
   (e) $\displaystyle\lim_{n\to\infty} \log_2\left(\frac{1}{\log_2(n)}\right)$

   (f) $\displaystyle\lim_{n\to\infty} \frac{n^n}{n!}$
10. Which of the three algorithms `BubbleSort`, `MergeSort` and `InsertionSort` is the fastest (i.e. needs the fewest operations) to sort a list of 5 elements? Which needs the fewest to sort a list of 10 elements? How about 100? How about for huge values of n?

Compsci 120	University of Auckland
Graphs	
2020	*Chapter 5*

Graphs

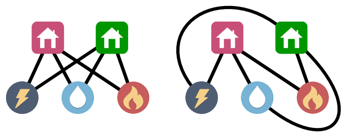
Exercise 5.1. *In the 1700's, the city of Königsberg was divided by the river Pregel into four parts: a northern region, a southern region, and two islands. These regions were connected by seven bridges, drawn in red in the map at right.*


A map of Königsberg, c. 1730.
Source: Wikipedia.

These bridges were particularly beautiful pieces of architecture, and so when people would offer tours of the city they would try to take tourists across each bridge to show it off. However, tour guides at the time noticed that no matter how they could construct their walks around the city, they'd always either miss a bridge or have to double a few bridges up: they could never find a "perfect" tour that crossed each bridge exactly once.

Can you find such a tour? That is: can you come up with a walk through the city that starts and ends at the same place, and walks over each bridge exactly once?

Exercise 5.2. *You're a civil engineer! Suppose that you have a set of buildings, and you want to hook up each of these buildings to the city's utilities, namely water, gas, and electricity.*

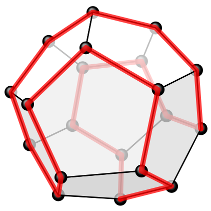


However, your city is very earthquake-prone. As a safety precaution, they've asked that no two utilities are allowed to vertically "cross over" each other: that is, you can't have water mains running beneath the gas mains, or gas pipes running above the electricity wires. So, for example, while the configuration drawn at left wouldn't work for two buildings, the right one would!

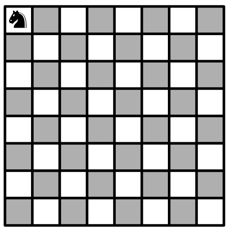
Can you connect three buildings to your utilities? Or is this impossible?

5.1 Graph Theory

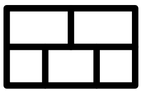
The field of **graph theory** originally started as a branch of "recreational mathematics," and specifically as the puzzle in Exercise 5.1. Until the 1900's, people viewed graph theory as a branch of mathematics used to solve riddles similar to the Königsburg bridge problem, such as the following:


A solution to the dodecahedron puzzle!
Source: Wikipedia

- Can you take a dodecahedron, start from one corner, and walk along the edges in a way that visits every other vertex exactly once and returns to where you start?



- Take a knight, and put it in the top-right-hand corner of an 8×8 chessboard. Can you move it around so that it visits every other square on the board exactly once, and returns to where it started?



- Take a box, divided into five rectangles. Can you draw a single continuous line that crosses each wall exactly once?

In recent years, however, mathematicians and computer scientists have transformed graph theory into one of the most applicable fields of mathematics in existence. Its power to describe networks has made it the perfect tool for studying many problems in the modern world; graphs can be used to model the internet, social networks, the spread of diseases through a population, travel, computer chip design, and countless other phenomena. Graphs are everywhere in the modern world, and their analysis is key to solving many of the most important problems of the 21st century.

In these notes, we'll start by building up some definitions that will let us talk about graphs and their properties; from there, we'll describe how to model various real-life objects with graphs, and then transition to solving a few real-life problems with graphs. This is (as with everything in this class) just the tip of the iceberg; if you want to see more, take papers like Compsci 225 and Maths 326!

We start with an intuitive definition of a graph:

Definition 5.1. *Intuitively, a **graph** is just a way of modeling a collection of objects and the connections between them.*

Example 5.1.

For example, all of the following objects can be thought of as graphs:

- Consider Facebook, or any other social network. You can think of any of these objects as consisting of two things: (1) the **people**, and (2) the **connections** (i.e. friendship, or following) between people.
- Think about the internet. You can model this as a collection of webpages, connected by links.
- Alternately, you can model the internet as a collection of computers, connected by cables/wireless signals.
- Look at all of New Zealand! You can model the country as a collection of cities, connected by roads.
- Alternately, you can choose to model New Zealand as a collection of regions, with two regions linked up if there's a direct flight from one to the other.

If Definition 5.1 above feels a bit too informal for you, here's a more precise way to describe a graph:

For simplicty's sake, we will use the word "graph" to refer to a simple undirected loopless graph, and assume that all graphs are simple undirected loopless graphs unless we explicitly say otherwise.

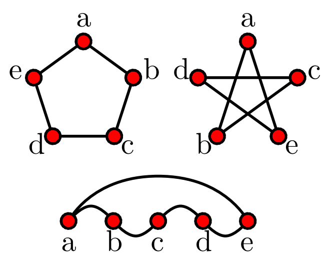
Definition 5.2. *A **simple undirected loopless graph** G consists of two things: a collection V of **vertices**, and another collection E of **edges**, which we think of as distinct unordered pairs of distinct elements in V. We think of the vertices in a graph as the objects we're studying, and the edges in a graph as a way to describe how those objects are connected.*

*To describe an edge connecting a pair of vertices a, b in our graph G, we use our set language from earlier and write this as $\{a, b\}$. We say that a and b are the **endpoints** of the edge $\{a, b\}$ when this happens.*

Example 5.2. The following describes a graph G:

- $V = \{a, b, c, d, e\}$
- $E = \big\{\{a,b\}, \{b,c\}, \{c,d\}, \{d,e\}, \{e,a\}\big\}$

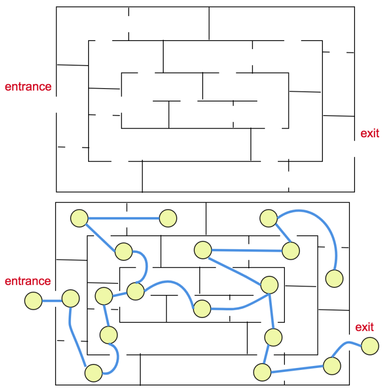
Given a graph $G = (V, E)$, we can **visualize** G by drawing its vertices as points on a piece of paper, and its edges as connections between those points.



Several ways of drawing the graph G defined above are drawn at right. Notice that there are many ways to draw the same graph! Also note that edges don't have to be straight lines, and that we allow them to cross.

In general, it is often much easier to describe a graph by drawing it rather than listing its vertices and edges one-by-one. Wherever we can, we'll describe graphs by drawing pictures; we encourage you to do the same!

Here's a few examples of graphs, described with this vertex+edge language:



Example 5.3.

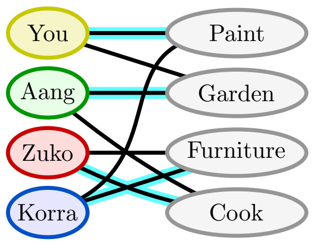
- We can represent a maze as a graph! Take the collection of "rooms" in our maze, and think of them as our vertices. Now, connect any two room-vertices with an edge if they are linked by a doorway.

  With this done, finding a way out of our maze is the same as finding a walk through our graph! This can often simplify things for us, as the graph visualization lets us ignore some of the more irrelevant information (i.e. right angles, walls without doors, etc) in the maze.

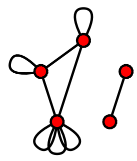
- You and your three flatmates (Aang, Korra, and Zuko) have just moved to a new house. You have a list of tasks to do over the weekend: you all want to paint the walls, move in your furniture, cook dinner, and do some gardening. Suppose that you have the following skills:

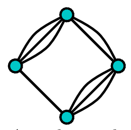
  – You really like to paint and garden.
  – Aang is a great cook who enjoys gardening.
  – Zuko is also a great cook who's strong (furniture.)
  – Korra is strong (furniture) and likes to paint.

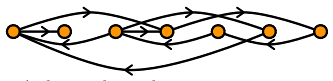
  You can visualize this with a graph! Make a vertex for each flatmate, and a vertex for each task. Then, connect flatmates to their preferred tasks with edges. With this, a "good" way to divide up tasks is to find a "matching:" that is, a way to pick out four edges in our graph, such that every person and every task is in exactly one edge (so that the work is divided!) Such a matching is highlighted in the graph at right.



Just as there are many different ways to model the connections between a set of objects, there are other notions of graphs beyond that of a **simple graph**. Here are some such definitions:


A simple graph with loops on five vertices.


A multigraph on four vertices.


A directed graph on seven vertices.

Definition 5.3. *A **simple graph with loops** is just like a simple graph G, except we no longer require the pairs of elements in E to be distinct; that is, we can have things like $\{v, v\} \in E$.*

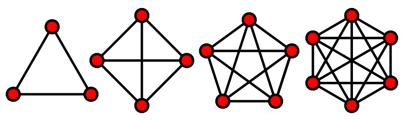
*A **multigraph** is a simple graph, except we allow ourselves to repeat edges in E multiple times; i.e. we could have three distinct edges $e_1, e_2, e_3 \in E$ with each equal to the same pair $\{x, y\}$.*

*A **directed graph** is a simple graph, except we think of our edges as ordered pairs: i.e. we have things like $x \to y \in E$, not $\{x, y\}$.*

You can mix-and-match these definitions: you can have a directed graph with loops, or a multigraph with loops but not directions, or pretty much anything else you'd want!

Remark 5.1. *In this course, we'll assume that the word **graph** means **simple undirected graph without loops** unless explicitly stated otherwise.*

There are a handful of graphs that come up frequently in computer science and mathematics, are are worth giving specific names. We describe a few of these graphs here:

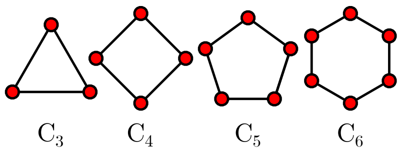

K_3 K_4 K_5 K_6

Definition 5.4. *The **complete graph on** n vertices K_n is defined for any positive integer n as follows: take n vertices. Now, take every possible pair of distinct vertices, and connect them with an edge! We draw several examples at right.*

In this sense, a complete graph is a graph in which we have as many edges as is possible for a graph on n vertices.

Definition 5.5. *The* ***complete bipartite graph on*** *m, n vertices $K_{m,n}$ is defined for any two positive integers m, n as follows: take $m + n$ vertices. Split them into two groups, one of size m and one of size n. Then, for every pair of vertices $\{a, b\}$ where a is from the m group and b is from the n group, connect a to b. These graphs have no edges connecting any two members from the "same" group, but have all of the possible "crossing" edges going from one group to the other.*

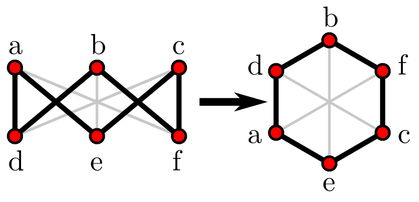
Definition 5.6. *The* ***cycle graph on*** *n vertices C_n is defined for any integer $n \geq 3$ as follows: take n vertices, and label them $1, 2, \ldots n$. Now, draw edges $\{1, 2\}$,$\{2, 3\}$, $\ldots$ until you get to the last edge $\{n-1, n\}$; then connect this up into a closed cycle by drawing $\{n, 1\}$ as an edge as well.*



To practice all of this graph theory language, let's try solving one of our exercises!

Answer to Exercise 5.2. You cannot solve the utility problem!

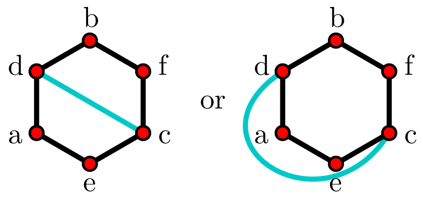
To see why, let's use the language of graph theory. Make each of our three buildings into vertices; as well, make each of the three utilities into vertices. If we connect each building to each utility, notice that we've created the graph $K_{3,3}$!



We claim that $K_{3,3}$ cannot be drawn without any edges crossing, and thus that this puzzle cannot be solved. To see why this is true, let's use the "contradiction" technique we used above.

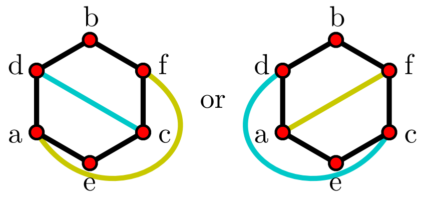
Think about what a solution would look like. In particular, notice that $K_{3,3}$ contains a "hexagon," i.e. a C_6, as drawn at right. Therefore, in any drawing of $K_{3,3}$, we will have to draw this cycle C_6 first.

Because this cycle is a closed loop, it separates space into an "inside" and an "outside." Therefore, if we are drawing $K_{3,3}$ without edges crossing, after drawing the C_6 part, any remaining edge will have to either be drawn entirely "inside" the C_6 or "outside" the C_6. That is, we can't have an edge cross from inside to outside or vice-versa, because that would involve us crossing over pre-existing edges.



Therefore, if we have a crossing-free drawing of $K_{3,3}$, after we draw the C_6 part of this graph, when we go to draw the $\{d, c\}$ edge we have two options: either draw this edge entirely on the inside of our C_6, or entirely on the outside.

If we draw this edge on the inside, then on the inside the vertices f and a are separated by this edge; therefore, to draw the edge $\{a, f\}$ we must go around the outside. Similarly, if we draw this edge on the outside, then a, f are separated from each other on any outside walk, and the edge $\{a, f\}$ must be drawn inside of the hexagon.



In either of these cases, notice that there is no walk that can be drawn from b to e on either the inside or the outside! Therefore we cannot draw our last edge $\{b, e\}$ without having a crossing, and thus have a contradiction to our claim that such a crossing-free drawing of $K_{3,3}$ was possible.

5.2 Graphs: Useful Concepts

In the above section, we saw how some of the language of graph theory could help us solve problems. In this section, we explore a number of other definitions and concepts that can come in handy!

Definition 5.7. *Take a graph G. We say that two vertices a, b in G are* **adjacent** *if the edge $\{a, b\}$ is in G. We say that a and b are* ***neighbors*** *if they are adjacent.*

Example 5.4. If we look at the flatmate graph from Example 5.3 above, we can see that the vertices "You" and "Paint" are adjacent, but that "You" and "Furniture" are not adjacent. Similarly, "Aang" and "Cook" are adjacent, but "Aang" and "Zuko" are **not** adjacent.

Notice that adjacency is just a property of individual edges! Even though we drew the Aang and Zuko vertices next to each other on our graph, we did not connect them with an edge, so they are not connected. As well, even though you can go from "You" to "Furniture" by using the multiple-edge walk

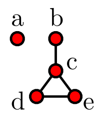
$$\text{You} \to \text{Paint} \to \text{Korra} \to \text{Furniture},$$

you cannot go from "You" to "Furniture" with a **single** edge, and so these vertices are not adjacent.

It is often useful to be able to refer to all of the neighbors of a vertex: in this graph, for instance, "Korra" is neighbors with "Paint" and "Furniture."

Definition 5.8. *Take a graph G, and a vertex v in G. We say that the* ***degree*** *of v, written $\deg(v)$, is the number of edges that contain v as an endpoint.*

Example 5.5. In the graph with red vertices drawn in the margins, the vertex a has degree 0, the vertex b has degree 1, the vertices d and e have degree 2, and the vertex c has degree 3.



Notice that a vertex can have degree 0: i.e. it is possible to have a vertex with no edges! This will often represent an "isolated" object in our graph: i.e. something without any connections to our wider graph.

Exercise 5.3. *A bit more practice with the concept of degree: convince yourself of the following claims!*

- *The degree of every vertex in K_n is $n - 1$.*
- *The only possible degrees in $K_{m,n}$ are m and n.*
- *The degree of every vertex in a cycle graph C_n is 2.*

To practice these concepts, let's try writing a few arguments:

Claim 5.1. *(The "degree-sum formula," or "handshaking theorem.") Take any graph G. Then, the sum of the degrees of all of the vertices in G is always two times the number of edges in G.*

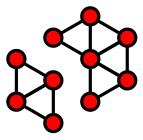
Proof. We can prove this by "counting" the number of times vertices in our graph show up as endpoints of edges in our graph. Notice that we could do this in two different ways:

1. On one hand, every edge has two endpoints. Therefore, the total number of times vertices are used as endpoints in our graph is simply twice the number of edges.
2. On the other hand, every endpoint is counted once when we calculate the degree of the corresponding vertex. Therefore, if we sum up the degrees of all of the vertices in our graph, this counts the total number of times vertices are used as endpoints in our graph.

However, both of these ways were counting the same thing! Therefore, we know that the answer we got from (1) must be the same as the answer we got in (2). In other words, the sum of degrees of vertices in our graph must be twice the number of edges, as claimed. □

Example 5.6. In the graph at right, we have four vertices of degree 2, five vertices of degree 3, and one vertex of degree 5. Therefore, the sum of degrees in this graph is $4 \cdot 2 + 5 \cdot 3 + 1 \cdot 5 = 28$.

Claim 5.1 says that this should be twice the number of edges; i.e. that this graph should have 14 edges. This is true: count them by hand!



This formula comes up often in graph theory, especially when trying to show that certain types of graphs are "impossible!" Consider the following result:

Claim 5.2. *You cannot have a graph on seven vertices in which the degree of every vertex is 3.*

Proof. Before reading this proof, try to show this yourself: that is, take pen and paper, and try to draw a graph on seven vertices where all of the degrees are 3!

In doing so, we make two claims:

- You won't succeed: you'll always wind up with one vertex forced to have degree 2 or 4 (or you'll accidentally make something that's not a simple graph by drawing multiple edges / loops.)
- It won't be obvious why this keeps failing: i.e. there will be a lot of different things you could have tried, and writing down a "proof" for why this is impossible may feel like it would involve a ton of tedious casework.

However, if we use the degree-sum formula, we claim that this problem is quite simple! We start by using a principle that's served us well in many prior problems: let's suppose that our claim is somehow false, and it is possible to have a graph G on seven vertices in which all degrees are 3.

The degree-sum formula, when applied to G, tells us that the sum of the degrees in G must be twice the number of edges. Because all of the degrees in G are 3 and there are seven vertices, we know that this degree sum is just $3 \cdot 7 = 21$.

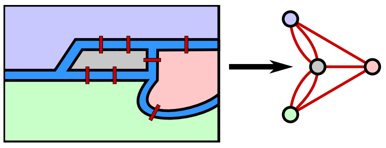
Therefore, we must have that two times the number of edges in our graph is 21. That is, we have 10.5 edges ... which is impossible, because edges come in whole-number quantities! (I.e. you either have an edge $\{a, b\}$ or you don't: there is no "half" of an edge $\boxed{\{a}$ in a simple graph.)

Therefore, such a graph G cannot exist, as the number of edges required for such a graph G is impossible to construct. In other words, no graphs exist on seven vertices in which all degrees are 3! ☐

5.3 Walks and connectedness

In several of the examples we looked at earlier, the idea of a **walk** through a graph was an incredibly useful one:

- In the "internet" graph, if we model this graph as a collection of computers connected by cables/wireless signals, in order for two people to communicate (i.e. send an email,) we have to find a way through this graph that links these two computers together.
- In the Königsberg bridge problem from Example 5.1, we can turn the city of Königsberg into a multigraph by making the four regions of the city into vertices, and connecting these vertices with edges.

  In this setting, our goal is to find a walk through this graph that uses every edge exactly once, and starts + ends at the same place.
- In the "maze" graph, we wanted to find a walk through our graph that starts at the entrance and leaves through the exit.

- In the New Zealand graph, where we had a bunch of cities as our vertices connected by roads, we often want to navigate from one city to the next. Doing so is finding a route through our graph!

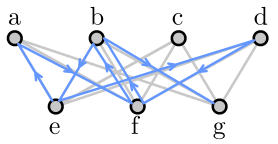
As such, we should come up with a definition for what a "walk" is in a graph:

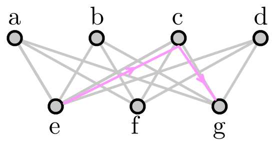
Definition 5.9. *In a graph G, we define a* **walk** *of* **length** *n as any sequence of n edges from G of the form*

$$\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}.$$

We say that this walk ***starts*** *at v_0 and* ***ends*** *at v_n.*

We say that a walk is a ***circuit*** *or* ***closed walk*** *if it starts where it ends; i.e. if $v_0 = v_n$.*





We say that a walk is a ***path*** *if it does not repeat any vertices, with the following exception: if the first and last vertex of path are the same and all of the others are distinct, we allow this to be a path as well. In this last case, we call our walk a* ***cycle****.*

We often describe a walk by just listing its vertices in order: i.e.

$$v_0 \to v_1 \to v_2 \to \ldots \to v_{n-1} \to v_n$$

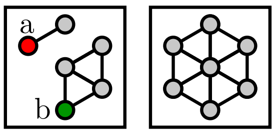
is a valid way to describe a walk.

Example 5.7. In the graph in the margins, the following sequences are walks from e to g:

- $e \to a \to f \to b \to e \to d \to f \to b \to g$
- $e \to c \to g$

Notice that walks can repeat edges and vertices!

A useful property that graphs can have, related to this concept of walks, is being **connected**:



Definition 5.10. *Given a graph G, we say that G is* ***connected*** *if for every pair of vertices a, b in G, there is a path from a to b in G.*

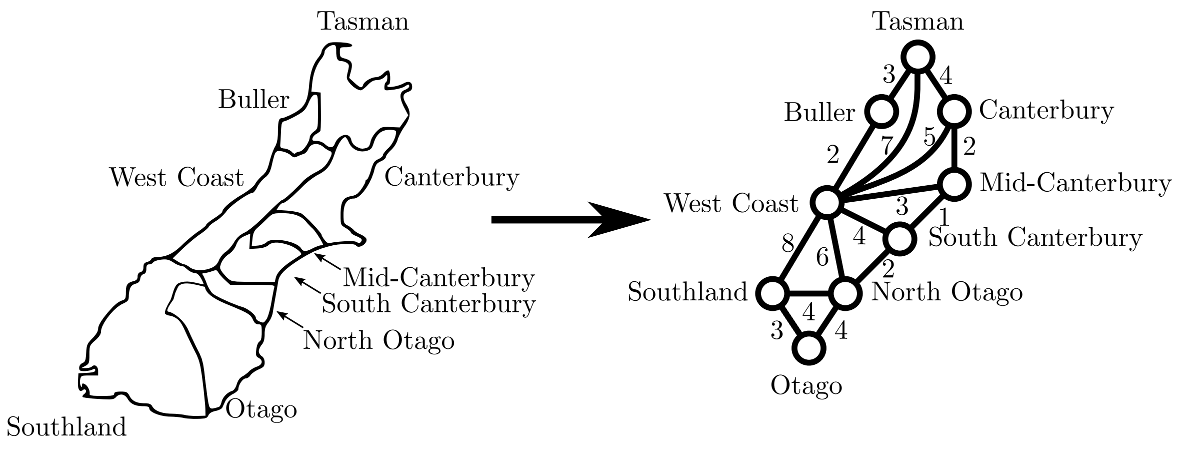
Example 5.8. The boxed graph at left is not connected, as there is no path from a to b in this graph. The one at right, however, is connected, as there is a path from any vertex to any other vertex.

In many applications, we'll want to find the **shortest path** between two objects: i.e. in a transit graph you'll want to find the path with the shortest possible length, or in the internet graph you'll want to find the path with the lowest total ping. To capture this idea, we'll often want to attach **weights** to the edges of our graph, to represent paths that are physically longer / more expensive to use / etc. We do this as follows:

Definition 5.11. *An* ***edge-weighted*** *graph G is a graph G along with a function f that assigns every edge a number.*

Example 5.9. A map of the South Island of New Zealand is drawn below. We can turn this into a graph by replacing each region with a vertex, and connecting two regions if they border each other.

With this done, we can turn this graph into a **weighted graph** by labelling each edge with the total amount of time it would take you to drive from the largest city in one region to the largest city in the next region. This gives us the graph below:

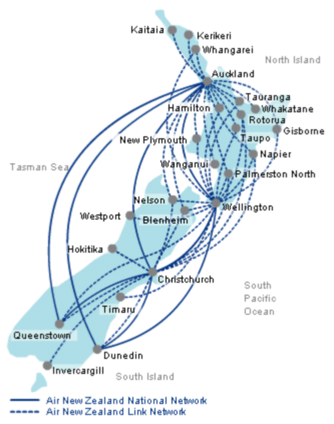


In graphs like this, we'll often solve problems like the following:

Example 5.10. Suppose that you're a **traveling salesman**. In particular, you're traveling the South Island, and trying to sell rugby tickets for nine rugby teams there (one for each region in the map above.)

You want to start and finish in Mid-Canterbury, and visit each other region exactly once to sell tickets in it. What circuit can you take through these cities that **minimizes** your total travel time, while still visiting each city **exactly once**?

Without knowing any mathematics, you'd probably guess that the shortest route is to just go around the perimeter of the island. Intuitively, at the least, this makes sense: avoiding the southern alps is probably a good way to save time!



Publicly-available map sourced from http://www.airlineroutemaps.com's Air New Zealand page.

In real life, however, maps can get a lot messier than this. Consider a map of all of the airports in the world, or even just in New Zealand (at right.) If you were an Air New Zealand representative and wanted to visit each airport, how would you do so in the shortest amount of time and still return home to Auckland?

In general: suppose you have n cities $C_1, \ldots C_n$ that you need to visit for work, and you're trying to come up with an order to visit them in that's the fastest. For each pair of cities $\{C_i, C_j\}$, assume that you know the time it takes to travel from C_i to C_j. How can you find the cheapest way to visit each city exactly once, so that you start and end at the same place?

These sorts of tasks are known as **traveling salesman problems**, and companies all over the world solve them daily to move pilots, cargo, and people to where they need to be. Given that it's a remarkably practical problem, you'd think that we'd have a good solution to this problem by now, right?

. . . not so much. Finding a "quick" solution (i.e. one with non-exponential runtime) to the traveling salesman problem is an **open problem** in theoretical computer science; if you could do this, you would solve a problem that's stumped mathematicians for nearly a century, advance mathematics and computer science into a new golden age, and quite likely go down in history as one of the greatest minds of the millenium .

So, uh, extra-credit problem.

This is a fancy way of saying "this problem is really hard." So: why mention it here? Well: in computer science in general, and graph theory in particular, we often find ourselves having to solve problems that don't have known good or efficient algorithms. Despite this, people expect us to find answers anyways: so it's useful to know how to find "good enough" solutions in cases like this!

For the traveling salesman problem, one brute-force approach you could use to find the answer could be coded like this:

Algorithm 5.9. *Init: Take our graph G, containing n vertices. Let s be the vertex we start and end at. Let c be a cost function, that given any edge $\{x, y\}$ in G outputs the cost of traveling along that edge.*

1. *Write down every possible walk in which we can list the n vertices of G, starting and ending at s.*
2. *For each walk $\{s, v_1\}, \{v_1, v_2\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, s\}$, calculate*

$$c(\{s, v_1\}) + c(\{v_1, v_2\}) + \ldots + c(\{v_n, s\}).$$

   Assume that $c(\{x, y\})$ is infinite if the edge doesn't exist (i.e. that it would take "forever" to travel along a walk that is impossible to travel along.)

3. *Output the smallest number/walk you find.*

Points in favor of this algorithm: it works! Also, it's not too hard to code (try it!)

Points against this algorithm: if you were trying to visit 25 cities in a week, it would take the world's fastest supercomputer over ten thousand years to answer your problem. (If you were trying to visit 75 cities, the heat death of the universe occurs before this algorithm is likely to terminate.)

This is because the algorithm needs us to consider every possible order of the n vertices in G to complete. There are $(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$ many ways in which we can order our n cities, and the factorial function grows incredibly quickly, as we saw before!

To see why, think about how you'd make an ordering of the cities. You'd start by choosing a city to travel to from s: there are $n-1$ choices here, as we can possibly go anywhere other than s. From there, we have $n-2$ choices for our second city, and then $n-3$ for our third city, and so on/so forth!

Another approach (which, as authors who would like to book their travel before the heat death of the universe, we are in favor of) is to use **randomness** to solve this problem! Consider the following algorithm:

Algorithm 5.10. *Init: Take our graph G, starting vertex s, and cost function c just like before.*

1. *Start from s and randomly choose a city we haven't visited, and then go to that city.*
2. *Keep randomly picking new cities until we've ran out of new choices, and then return to s.*
3. *Calculate the total cost of that path.*
4. *Run this process, say, ten thousand times (which, while large, will be much smaller than $n!$ for almost all values of n that you'll run into.)*
5. *Output the smallest number/path you find.*

Points against this algorithm: strictly speaking, it probably won't work. That is: we're just repeatedly randomly picking path and measuring their length. There's no guarantee that we'll ever pick the "shortest" path!

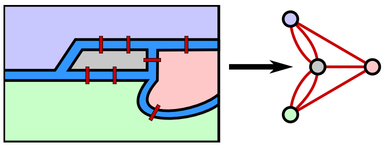
Points in favor of this algorithm: it's easy to code, it's really fast, and if you only care about just getting close-ish to the right answer it's actually[1] not too bad in many situations!

In the long run, it's probably better if your phone gives you slightly suboptimal directions in a second rather than taking two years to find the absolute best walk to the Countdown, so in general this is probably a better way to go. But in certain small situations (or times when you randomly have a supercomputer at hand) brute-force can also be the way to go: it really depends on what you're trying to solve!

To close this section, let's use our language of walks to answer our last remaining exercise:

[1]In particular there are lots of tweaks you can apply here to make this pretty decent in most cases, while still keeping it fast.

Answer to Exercise 5.1. It is impossible to construct such a walk! To see why, let's use graph theory and reduce our city of Königsberg to a multigraph, as done earlier.



With this done, we can see that each vertex in this graph has **odd** degree: the three outer ones have degree 3, and the inner one has degree 5.

We claim that this means that we cannot have a circuit (i.e. a walk that starts and ends at the same place) that only uses every edge once!

To see why, we use the "assume not" / contradiction technique that has often worked for us in the past. Suppose that this graph **did** have a circuit that used every edge exactly once. Describe this circuit in general as follows:

$$\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_0\}.$$

Pick any vertex x in our graph. Notice that each time x comes up in the above circuit, it does so twice: if $x = v_i$ for some i, it shows up in both $\{v_{i-1}, v_i\}$ and $\{v_i, v_{i+1}\}$. You can think of this as saying that each time our circuit "enters" a vertex along some edge, it must "leave" it along another edge!

As a result, any vertex x shows up an even number of times in the circuit we've came up with here. As well, we assumed that this circuit contains every edge exactly once. Therefore, every vertex x shows up in an even number of edges in our graph!

That is, $\deg(x)$ is even for every vertex x.

However, we know that our graph has odd-degree vertices. This is a contradiction! Therefore, we have shown that it is impossible to solve this puzzle.

5.4 Practice Problems

1. (-) Can you find a graph on 9 vertices in which all vertices have degree 1? How about 10?
2. Prove or disprove: if G is a graph on an odd number of vertices, then the degree of every vertex in G must be even.
3. Show that if G is a graph on n vertices, then G contains at most $\frac{n(n-1)}{2}$ edges.
4. (+) Show that if G is a graph on 10 vertices with at least 37 edges, then G must be connected.
5. (-) Find the shortest possible length of a circuit in the South Island map from Example 5.9.
6. Suppose you used Algorithm 5.9 to solve problem 1. How many iterations would it take to find the answer?
7. Draw a map for where you grew up. Label your home, school, local grocery store, and a couple of your favorite places to visit outside of home. Use Google Maps or something similar to find the distances between these things. What's the shortest walk that visits all of them, starting and ending from home?
8. (+) Can you draw K_5 without having any edges cross? Either do so, or explain why this is not possible.

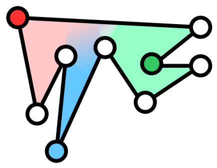
Compsci 120 University of Auckland

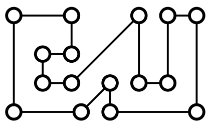
Trees

2020 *Chapter 6*

Exercise 6.1. *A graph G on $2n$ vertices is said to have "doubled degrees" if it has exactly 2 vertices with degree k, for every $k \in \{1, 2, \ldots n\}$. For example, the graph drawn at right has doubled degrees.*

How many trees exist with "doubled degrees?"


A gallery guarded by 3 cameras


Your gallery!

Exercise 6.2. *You're a curator of a large modern art museum! Because your museum is particularly "edgy," the room in which you're displaying your artwork is a very strange-looking polygon (see margins.)*

You want to install $360°$ cameras in the corners of your gallery, in such a way that your cameras see the entire room. What is the smallest number of cameras you need to install?

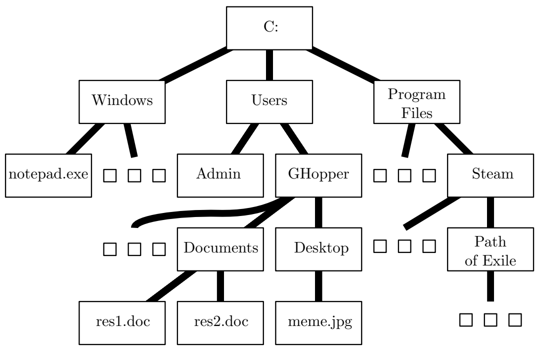
In general, what's the largest number of cameras you could need for an art gallery that is a n-sided polygon?

6.1 Trees

In our last section, we saw that the language of graph theory could be used to describe tons of real-life objects: the internet, transportation, social networks, tasks, and many other things! Even your computer (a particularly relevant thing to consider in a computer science class) can be described as a graph:

- Vertices: all of the files and folders in your computer.
- Edges: Draw an edge from a file or folder to every object it contains.

If you draw this out, you'd get something similar to the drawing below:

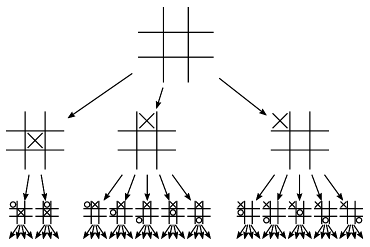


This graph represents the **file system** for your computer, and is extremely useful for organizing files: imagine trying to find a document if literally every file on your computer had to live on your desktop, for instance!

This graph has a particularly useful structure: starting from $\boxed{\text{C:}}$, there's always exactly **one** way to get to any other file or folder if you don't allow backtracking. That is: there are no files you can't get to by starting from your root and working your way down, and also there are no files that you can get to in multiple different ways! This is a very nice property for a file system to have: you want to be able to navigate to every file in some way, and it's very nice to know that files in different places are different (imagine deleting a file from your desktop and having all copies of it disappear in other places!)

We call graphs with the structural property described above **trees**. Trees come up all the time in real life:

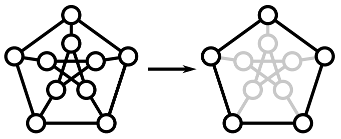
- PDF documents (like the one you're reading right now!) are tree-based formats. Every PDF has a root note, followed by various sections, each of which contains various subsections.
- In genealogy and genetics, people study **family trees**: i.e. take your great-grandmother, all of her children, all of her children's children, and so on/so forth until you're out of relatives. This is a tree, as starting from your great-grandmother there *should* only be one way to get to any relative.
- Given any game (e.g. chess, or tic-tac-toe, or Starcraft), you can build a **decision tree** to model possible outcomes as the game progresses. To do so, make a vertex for the starting state. Then make a vertex for every possible move player 1 could make, and connect the starting state to all of these. For each of those states, make a vertex for every response player 2 could make, and connect those states up as well; doing this for all possible moves generates a decision tree, which you can use to win!



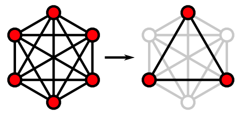
In short: they're useful!

To define what a tree is, we first need to introduce a useful concept from graph theory that we didn't have time to discuss last chapter:

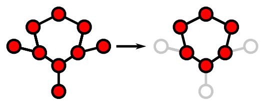
Definition 6.1. *A graph G has another graph H as a* ***subgraph*** *if H is "contained within" G. In other words, if you can take G and remove vertices and/or edges from it until you get the graph H, then H is a subgraph of G.*



For example, the graph at right has C_5 as a subgraph, because we can delete the "inside" vertices and edges to have just a C_5 left over.

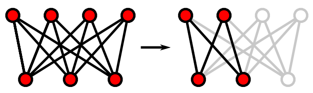


Note that any graph G is "trivially" a subgraph of itself, as we can just delete "nothing" from G and have G left over.



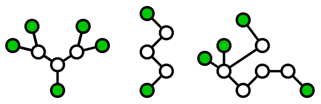
With this stated, we can define a tree as follows:

Definition 6.2. *A* ***tree*** *is a graph T that is connected and has no cycle graph C_n as a subgraph.*



For example, the three graphs in the margins are not trees, as each of them has a cycle graph of some length as a subgraph.

However, the three graphs below are all trees:



Happy little trees.

We call vertices of degree 1 in a tree the **leaves** of the tree. For example, the leaves of the trees above are colored green.

To get a bit of practice with these ideas, let's prove a straightforward claim about trees:

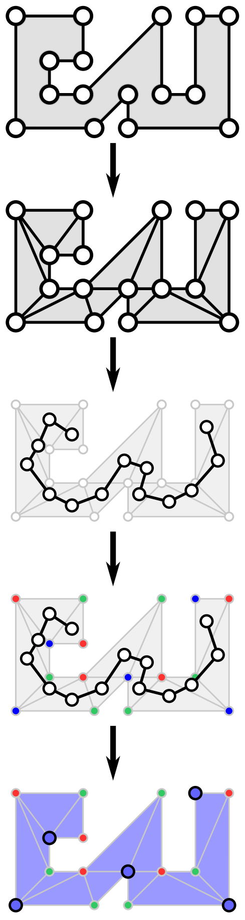
Theorem 6.1. *If T is a tree containing at least one edge, then T has at least two leaves.*

Proof. Consider the following process for generating a path in T:

Algorithm 6.11.

1. *Choose any edge $e = \{x_0, y_0\}$ in G.*
2. *Starting from $i = 0$, repeatedly do the following: if x_i has degree ≥ 2, then pick a new edge $\{x_i, x_{i+1}\}$ leaving x_i. Because T is a tree, x_{i+1} is not equal to any of our previously-chosen vertices (if it was, then we'd have created a cycle.) Stop when x_i eventually has degree 1.*
3. *Starting from $i = 0$, do the same thing for y_i.*

Notice that this process must eventually stop: on a tree with n vertices, we can only put n vertices in our path because the "no cycle" property stops us from repeating vertices. When it stops, the endpoints of the path generated are both leaves because this is the only way we stop this process. Therefore, this process eventually finds two leaves in any tree! □



As a bit of extra practice, let's try to use our tree language to sketch a solution to our second exercise:

Answer to Exercise 6.2. It turns out that you can guard any n-sided polygon (without any holes, and where all sides are straight) with at most $\lfloor \frac{n}{3} \rfloor$ cameras! To do so, use the following process:

- Take your n-sided polygon. By connecting opposite vertices, divide it up into triangles.
- Turn this into a graph: think of each triangle as a vertex, and connect two triangles with an edge when they share a side.
- This graph is a tree! (Why? Justify this to yourself.)
- Use this tree structure to do the following:
  - Take any triangle. Color its 3 vertices red, blue, and green.
  - Now, go to any triangle that shares a boundary with that colored triangle. It will have 2 of its three vertices given colors. Give its third vertex the color it's currently missing.
  - Repeat this process! It never runs into conflicts, because our graph is a tree (and so we don't have cycles.)
- Result of the above: every triangle has one red vertex, one blue vertex, and one green vertex.
- Put a camera on the least-used color! This needs at most $n/3$ rounded down cameras, as we're using the least popular of three colors. It also guards everything, as a camera sees everything in each triangle it's in!

6.2 Useful Results on Trees

One particularly useful thing about trees is that they can be defined in many different ways! In the section above, we define a tree as a **connected** graph with no **cycles**. However, we have two other properties that also characterize when a graph is a tree:

Recall from Claim that that two statements are **equivalent** if they hold in precisely the same situations: i.e. whenever one is true, the other is true, and vice-versa.

Theorem 6.2.

$$\boxed{\textit{T is a tree}} \quad \textit{is equivalent to} \quad \boxed{\begin{array}{c}\textit{there is exactly one}\\ \textit{path between any two}\\ \textit{vertices in } T.\end{array}}$$

Theorem 6.3.

$$\boxed{\textit{T is a tree}} \quad \textit{is equivalent to} \quad \boxed{\begin{array}{c}\textit{T is connected}\\ \textit{and has } n-1 \textit{ edges}\end{array}}$$

We prove the first of these theorems here:

Proof of Theorem 6.2. Because we're proving that these two statements are *equivalent*, we need to show that if either of them is true, then the other statement follows. That is, if you wanted to show that "attending office hours" and "getting an A+ in Compsci 120" were equivalent things, you wouldn't be satisfied if I said "everyone who got an A+ in Compsci 120 attended office hours:" you'd also want to know whether "everyone who attended office hours got an A+"!

As such, this proof needs to go in 2 steps:

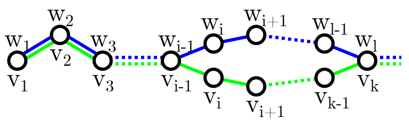
1. First, we need to show that if T is a tree, then there's a unique path between any two vertices in T.
2. Then, we need to show that if there's a unique path between any two vertices in T, then T is a tree.

We do each of these one-by-one:

1. Because T is a tree, by definition we know that T is connected. By the definition of connected, we know that for any two vertices x, y there is at least one path that goes from x to y. To complete our proof, then, we just need to show that **there aren't multiple paths** between any two vertices.

   To see why two distinct paths is impossible, we proceed by contradiction: i.e. we suppose that we're wrong, and that it *is* somehow possible for us to have two different paths linking a pair of vertices.

   Let's give those vertices and paths names: that is, let's assume that there are vertices x, y linked by two different paths $P_1 = \{v_1 = x, v_2\}, \{v_2, v_3\}, \ldots \{v_{n-1}, y\}$ and $P_2 = \{w_1 = x, w_2\}, \{w_2, w_3\}, \ldots \{w_{m-1}, y\}$. Because these two paths are different, there must be some value i such that $v_i \neq w_i$. Let i be the smallest such value, so that these paths agree at $v_i = w_i$ and diverge immediately afterwards.



   These two paths must eventually meet back up, as they end at the same vertex y. Let k, l be the two smallest values greater than i such that $v_k = w_l$. Notice that this means that all of the vertices $v_{i-1}, v_i, v_{i+1}, \ldots v_{k-1}, v_k, w_{l-1}, w_{l-2} \ldots, w_i$ are distinct (as otherwise we could have picked even smaller values at which these paths met back up.)

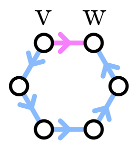
   Now, look at the walk formed by starting P_1 at v_{i-1}, proceeding until v_k, and then taking P_2 backwards from w_l until w_{i-1}. This walk repeats no vertices other than the starting and ending one, by construction. Therefore it is a cycle!

   But we are in a tree, and trees do not contain cycles. Therefore this is a *contradiction* to our assumption that we had two distinct paths. In other words, our assumption that two paths could exist was false, and we must have exactly one path, as claimed!

2. More-or-less, we can just reverse the argument above!

   That is: if T has our unique path property, then every two vertices in T are connected by a path, and so T is connected.

   To see why T cannot have any cycles: simply notice if T *did* contain a cycle, then it would give us two different paths between two vertices: you could go one way or the other around the cycle! Therefore, our unique path property stops us from having cycles, and thus means that T is a tree.



□

This result should help us understand how our "trees don't have cycle" property connects to the "there's exactly one path from $\boxed{\text{C:}}$ to any other file" property that made our filesystem example so useful!

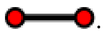
The second of these results requires induction, and so we'll delay its proof until Section 7.9. It's quite useful, though, and worth knowing even if we can't prove it yet! For example, it lets us solve one of our exercises:

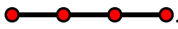
Answer to Exercise 6.1. Let T be a tree on $2n$ vertices with the "doubled degrees" property. Notice that if a graph T on $2n$ vertices has the "doubled degree" property, then the sum of the degrees in T is $1+1+\ldots+n+n = 2(1+2+\ldots+n) = 2\frac{n^2+n}{2} = n^2+n$.

As well, the degree-sum formula from our graph theory chapter tells us that the sum of the degrees in T is twice the number of edges. Therefore, we have that $n^2+n = 2E$.

Finally, because T is a tree, we know that it has one less edge than it has vertices (i.e. it has $2n-1$ edges, because G is on $2n$ vertices.)

Combining this all together tells us that $n^2+n = 2(2n-1) = 4n-2$; i.e. $n^2-3n+2=0$; i.e. $(n-1)(n-2)=0$, i.e. $n=1$ or $n=2$. In other words, if T is a tree with the doubled degrees property, then T is either a tree on 2 or 4 vertices.

For $n=1$, the "doubled degree" property would tell us that T should be a two-vertex graph with two vertices of degree 1. There is exactly one tree of this form, namely .

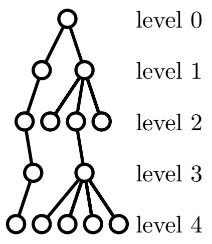
For $n=2$, the "doubled degree" property would tell us that T should be a four-vertex graph with two vertices of degree 1 and two of degree 2. There is also exactly one tree of this form, namely .

6.3 Rooted Trees

In many of the examples we looked at earlier (file systems, genealogy) it is natural to think of one vertex in our tree as the start, or **root** of our tree, and of all of our other vertices as "descending" from that root vertex. We formalize this idea with the concept of a **rooted tree**, defined here:

Definition 6.3. *We say that a **rooted tree** is any tree in which we designate one vertex r to be the "root" of the tree.*

Given a rooted tree, we can draw it as follows:



Algorithm 6.12.

0: At the top of the page, draw the root vertex r. We think of this as "level 0" of the tree.

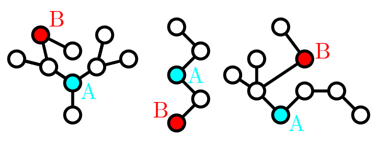
1: Below this vertex, draw all vertices adjacent to r along with those edges. Call this "level 1."

2: Below these vertices, draw all vertices that are adjacent to vertices in level 1 that have not already been drawn. Call this "level 2."

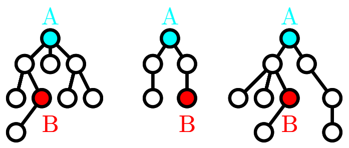
...

k: In general, if level $k-1$ has been drawn, draw all vertices that are adjacent to vertices in level $k-1$ that have not already been drawn. Call this "level k."

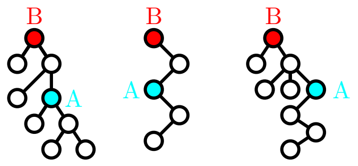
Keep doing this until you run out of vertices to draw!



Example 6.1. Three trees are drawn in the margins. Below, we draw each of them with vertex A chosen as the root:



We do this again,but now with vertex B chosen as the root:



The following terms are useful when discussing rooted trees:

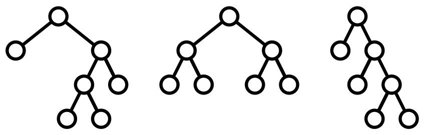
Definition 6.4. *We say that the **children** of a vertex v are all of the neighbors of v at the level directly below v, and the **parent** of v is the neighbor of v at the level directly above v. The **height** of a rooted tree is the largest level index created when drawing the graph as above.*

A particularly useful tree in computer science is a **binary tree**:

Definition 6.5. *A **binary tree** is a rooted tree in which every vertex has either no children, one child, or 2 children. A binary tree is called **full** if every vertex only ever has no children or two children.*

You can generalize this to m-ary trees for any m, by changing the restriction here to ask that every vertex has at most m children.

Example 6.2. Three binary trees are drawn below.



To finish out our chapter and practice working with this concept, let's study a quick result about binary trees:

Exercise 6.3. *Suppose that T is a full binary (i.e. 2-ary) tree with 100 leaves. How many vertices does T have in total?*

Answer to Exercise 6.3. There are three kinds of vertices in T:

- The root vertex r. Because this is a full binary tree that contains more than one vertex, r must have exactly two children, and thus has degree 2.
- All of the other parent vertices. Each of these have two children because we're a full binary tree, and exactly one parent by (c) + the fact that they're not the root. So these all have degree 3.
- All of the leaves, which have degree 1.

Suppose that there are p parent vertices in our graph; then the sum of degrees in this graph is $2 + 3p + 100$. As shown in class, the sum of degrees in any graph is twice the number of edges. Because this is a tree on $1 + p + 100$ vertices (100 leaves, one root, and p parents), this is therefore equal to $2p + 200$, as any tree has one less edge than it has vertices.

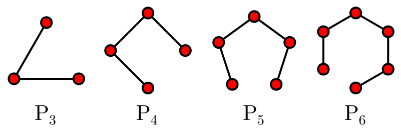
So: $2+3p+100 = 2p+200$ implies that $p = 98$, and therefore that we have $\boxed{199}$ vertices in total!

6.4 Practice Problems

1. (-) In a rooted tree on n vertices, what is the maximum number of children a vertex can have?
2. Show that in a rooted tree, no vertex has two distinct parents.
3. Suppose that T is a full ternary (i.e. 3-ary) tree with exactly 99 leaves. How many vertices in total does T have?
4. Suppose that G is a graph with the following two properties:
   - G is connected.
   - If we delete any edge from G, G is no longer connected.

   Show that G is a tree.
5. Suppose that T is a rooted full binary tree on 99 vertices. What is the maximum height of T? What is the minimum height of T? Justify your claims.
6. (+) A *graceful labeling* of a graph with E edges is a labeling $l(v)$ of its vertices with distinct integers from the set $\{0, 1, 2, \ldots E\}$, such that each edge $\{u, v\}$ is uniquely determined by the difference $|l(u) - l(v)|$.

   Let P_n denote the n-vertex **path graph**, formed by drawing n vertices $v_1, \ldots v_n$ in a row and connecting each v_i to v_{i+1} (drawn in the margins.)

   Show that each P_n tree is graceful.
7. (+) A *caterpillar tree* is a tree such that deleting all of its leaves leaves us with a single path (i.e. they kinda look like caterpillars.)

   Show that all caterpillar trees are graceful.
8. (++) Show that all trees are graceful.

Compsci 120 University of Auckland

Proofs

2020 *Chapter 7*

Exercise 7.1. *You're about to leave on holiday, but you forgot to pack socks! You've ran back to your room, but the light's burnt out, so you can't see the colours of your socks.*

You know that in your sock drawer that there are ten pairs of green socks, ten pairs of black socks, and eleven pairs of blue socks (all mixed up.)

How many of your socks do you need to take before you can be sure you've grabbed at least one matching pair?

Exercise 7.2. *You're a mad scientist! You've conducted an experiment on yourself to get superpowers. It worked, but to keep the powers you need to take two different tablets each day; if you forget one, or take more than one of either type, you'll, um, explode.*

Unfortunately, they look completely identical, and you've just dropped your last two days of supply (four tablets) on the floor.

What can you do?

7.1 Proofs: Motivation and Fundamentals

Throughout this coursebook, we've stressed the importance of clear, logical explanations for *why* things are true. In our previous chapters, we motivated the need for these arguments in a number of ways:

- In our answer to Exercise 1.1, we said that making a good, logical argument is often a useful skill in the workplace! In real life, you will often have to work for people who aren't particularly "tech-y" and will expect you to be able to do literally impossible tasks. Being able to give a clear and patient explanation for why something **can never happen** is a good skill to have!
- In our argument for Claim 1.1, we explained why an argument that just checks a few hundred or even a few thousand cases is often not enough in mathematics and computer science. As we saw then, there are tons of examples of claims that are true for all of the small values you'd like to check, but that suddenly become completely false when you get to large values (i.e. the ones that you could encounter when running code!)
- Finally, in chapter 4 we studied **algorithms**, step-by-step processes that we could easily turn into computer programs. When doing so, we often added in discussions about why these algorithms were "guaranteed" to work or why these algorithms would never "crash;" these arguments made it so that we could trust these processes to always work.

  This is something that you do in real life as a computer scientist all the time! Good code involves carefully thinking about all of the possible inputs you could be given, and for each case ensuring that your code is well-behaved. Writing such code and documenting it in a way that others can understand is a key part of being a professional programmer!

These motivations aren't just hypotheticals! Here's a pair of stories that illustrate why knowing how to make logical arguments is a useful skill:

Story 7.1. Janelle Shane is a researcher in optics who works with neural networks and machine learning. Roughly speaking, the way that a neural network works is the following:

- Take a bunch of examples of the thing you want the neural network to recognize, as well as a bunch of nonexamples.
- "Show" the neural network these examples and nonexamples.
- The neural network will then come up with a set of rules that it believes describes what it means for

"Show" is hard to define in words, but there are some great YouTube videos: see SethBling's Mar I/O videos for a fun and accessible introduction!

People are often tempted to just use the results of a neural network directly, without checking whether its discovered rules make sense. Doing so, as Janelle notes, leads to some fascinatingly weird behaviour:

- "There was an algorithm that was supposed to sort a list of numbers. Instead, it learned to delete the list, so that it was no longer technically unsorted."
- " In 1997, some programmers built algorithms that could play tic-tac-toe remotely against each other on an infinitely large board. One programmer, rather than designing their algorithms strategy, let it evolve its own approach. Surprisingly, the algorithm suddenly began winning all its games. It turned out that the algorithms strategy was to place its move very, very far away, so that when its opponents computer tried to simulate the new greatly-expanded board, the huge gameboard would cause it to run out of memory and crash, forfeiting the game."
- "An algorithm that was supposed to figure out how to apply a minimum force to a plane landing on an aircraft carrier. Instead, it discovered that if it applied a *huge* force, it would overflow the programs memory and would register instead as a very *small* force. The pilot would die but, hey, perfect score."

In short: just because something works for a bunch of examples doesn't mean it's good!

Story 7.2. A somewhat darker story on the importance of being able to read and understand proofs comes from the NSA, and something called a Dual Elliptic Curve Deterministic Random Bit Generator. This was an algorithm, designed by the NSA (a USA security agency,) that they claimed was a cryptographically secure way to generate random numbers.

See the New York Times for an article summarizing the scandal.

Also, check out Kleptography: Using Cryptography Against Cryptography and Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator , if you'd like to read through some research papers describing the NSA's algorithm/its weaknesses.

However, this algorithm was one that the NSA had built a "backdoor" into. That is, they designed the algorithm around certain secret values so that anyone with knowledge of those values (i.e. the NSA) could predict the randomly-generated numbers with a higher-than-normal degree of accuracy and thereby defeat cryptographic systems using this algorithm.

The NSA managed to get their algorithm used as a "standard" for over seven years. However, many mathematicians and computer scientists were suspicious of the NSA's algorithm from the very start, in large part **because it was not something that was proven to work**! Their research led to the eventual revocation of the NSA's algorithm as a standard.

So: we have some motivation for *why* we would want to write clear, logical arguments. The next question for us, then, is what counts as a valid argument?

Every major field of study in academia, roughly speaking, has a way of "showing" that something is true. In English, if you wanted to argue that the whale in Melville's Moby Dick was intrinsically tied up with mortality, you would write an essay that quoted Melville's story alongside some of of his other writings and perhaps some contemporary literature, and logically argue (using these quotations as "evidence") that your claim holds. Similarly, if you were a physicist and you wanted to show that the speed of light is roughly $3.0 \cdot 10^8$ meters per second, you'd set up a series of experiments, collect data, and see if it supports your claim.

In mathematics, a **proof** is an **argument** that mathematicians use to show that something is true. However, the concepts of "argument" and "truth" aren't quite as precise as you might like; certainly, you've had lots of "arguments" with siblings or classmates that haven't proven something is true!

In mathematics, the same sort of thing happens: there are many arguments that (to an outsider) look like a convincing reason for why something is true, but fail to live up to the standards of a mathematician. In Chapter 1, we already studied a pair of "failed" proofs: namely, our first attempts at proving Claim 1.1 and Exercise 1.1. We said that these arguments failed because they did not work in **general**: that is, they only considered a few cases, and did not consider **all** of the possible ways to put dominoes on a chessboard, or to pick a pair of integers.

This, however, is not the only way in which a proof might fail us! Here's another dodgy proof:

Claim 7.1. *Given any two nonnegative real numbers x, y, we have $\frac{x+y}{2} \geq \sqrt{xy}$.*

"Bad" proof:

$$\begin{aligned}
\sqrt{xy} &\leq \frac{x+y}{2}\\
xy &\leq \frac{(x+y)^2}{4}\\
4xy &\leq (x+y)^2\\
4xy &\leq x^2 + 2xy + y^2\\
0 &\leq x^2 - 2xy + y^2\\
0 &\leq (x-y)^2.
\end{aligned}$$

□

A defense of the "bad" proof: We're not using examples; we're working in general! Also, we *totally* showed that this claim is true: after all, we started with our claim and turned it into a true thing!

Why this proof is not acceptable in mathematics:

- We have no idea what x and y are! In particular, by plugging in some sample values of x and y, we can see that this is sometimes true and sometimes false: for $x = 1, y = 4$ we do indeed have $\sqrt{xy} = \sqrt{4} = 2 \leq \frac{1+4}{2} = 2.5$, but for $x = -1, y = -1$ the claim $\sqrt{(-1)\cdot(-1)} \leq \frac{-1-1}{2}$ is very false, as $-1 \nleq 1$! So, to do anything here, we first need to know what x and y **are**. That is: we need to define what set x, y come from!
- This proof is "backwards:" that is, it starts by assuming our claim is true, and from there gets to a true statement. This is not a logically sound way to make an argument! For example, if we assume that 1=2, we can easily deduce a true statement by multiplying both sides by 0:

$$\begin{aligned}
&1 = 2\\
\Rightarrow &0 \cdot 1 = 0 \cdot 2\\
\Rightarrow &0 = 0.
\end{aligned}$$

This doesn't prove that 1=2, though! As we said above, proofs need to **start** with true things, and then through argument **get to** what you're trying to show.

- Finally, this proof has no words! This flaw in some sense is why the other two flaws could exist: if you had to write out in words what x and y were, and how you went from one line to the next, it would probably become clear that this proof was written backwards and also that we have to be careful with what x, y are allowed to be.

This sort of thing is often easy to fix, though! If your proof is "backwards," simply try starting from the end and reasoning your way backwards to the start. If your logic was flawed, somewhere along the way you'll encounter a nonreversible step.

For example, if we tried to reverse our proof that $1 = 2$, we could go from $0 = 0$ to $0 \cdot 1 = 0 \cdot 2$, but would see that we can't "divide by 0" to get to the desired conclusion (and thus that this doesn't work.)

With this in mind, let's try a "fixed" version of this proof:

Theorem 7.1. *(The arithmetic mean-geometric mean inequality.) For any two nonnegative real numbers x, y, we have that the geometric mean of x and y is less than or equal to the arithmetic mean of x and y: in other words, we have that*

$$\sqrt{xy} \leq \frac{x+y}{2}.$$

Proof. Take any pair of nonnegative real numbers x, y. We know that any squared real number is nonnegative: so, in specific, we have that the square of $x - y$, $(x-y)^2$ is nonnegative. If we take the equation $0 \leq (x-y)^2$ and perform some algebraic manipulations, we can deduce that

$$\begin{aligned} 0 &\leq (x-y)^2 \\ \Rightarrow 0 &\leq x^2 - 2xy + y^2 \\ \Rightarrow 4xy &\leq x^2 + 2xy + y^2 \\ \Rightarrow 4xy &\leq (x+y)^2 \\ \Rightarrow xy &\leq \frac{(x+y)^2}{4}. \end{aligned}$$

Because x and y are both nonnegative, we can take square roots of both sides to get

$$\sqrt{xy} \leq \frac{|x+y|}{2}.$$

Again, because both x and y are nonnegative, we can also remove the absolute-value signs on the sum $x + y$, which gives us

$$\sqrt{xy} \leq \frac{x+y}{2},$$

which is what we wanted to prove. □

Much better! This proof doesn't have logical flaws, it's easier to read, and we've justified all of our steps so that even a skeptical reader would believe us.

7.2 Direct Proofs

In our past chapters, we've seen a number of useful techniques for writing such arguments while studying new mathematical concepts. In this chapter, we're reversing this structure a bit: instead of focusing on new mathematics, we're going to focus our sections on **new argument techniques**. By doing this, we're hoping that the arguments you've been reading in the past few chapters will become ones that you're comfortable with writing and reading on your own!

The proof we just wrote serves as a nice example of the first proof technique we'll study in this class: the idea of a **direct proof**. To prove that a given claim is true, the most straightforward path we've used in this class has been the following:

- Write down things that you know are true that relate to your claim. This typically includes the definitions of any terms referred to in the definition, any results from class or the tutorials/assignments that look related, and maybe some fundamental facts you know entering this class about numbers.
- Combine those things by using logic or algebra to create more things you know are true.
- Keep doing this until you get to the claim!

A particularly common form of direct proof comes up when people want to prove a statement of the form "if A holds, then B must follow" for two propositions A and B (or equivalently, "A implies B," which we write in symbols as $A \Rightarrow B$.)

To write a direct proof of such a statement, we proceed as before, but *also* throw in the assumption that A holds! That is, to prove "A implies B," we assume that A is true, and try to combine this assumption with other known true things to deduce that B is true. (Logically speaking, this is because $A \Rightarrow B$ holds as long as we're never in the situation where A is true and B is false. Therefore, if we can show that A being true forces B to also be true, then we know that our claim must hold!)

We illustrate this with a pair of examples here:

Claim 7.2. *If n is an odd integer, then n^2 can be written as a multiple of 4 plus one.*

Proof. We start by "assuming" the part by the "if:" that is, we assume that n is an odd integer. By definition, this means that we can write $n = 2k + 1$ for some other integer k.

We seek to study n^2. By our observation above, this is just $(2k + 1)^2 = 4k^2 + 4k + 1 = 4(k^2 + k) + 1$. This is a multiple of 4 plus 1, as claimed! Therefore we have completed our proof. ☐

Claim 7.3. *If G is a graph, then G must have an even number of vertices with odd degrees; that is, it is impossible to have a graph G with an odd number of vertices with odd degrees.*

Proof. We start this proof by thinking about all of the facts that we know about graphs and degrees. There's one result that should immediately jump to mind, namely the **degree-sum formula**: for any graph G,

$$\boxed{\begin{array}{c}\text{The sum of the degrees}\\ \text{of the vertices in } G\end{array}} = \boxed{\begin{array}{c}\text{Twice the number}\\ \text{of the edges in } G\end{array}}$$

Let's use this result! Specifically: in this problem, we're studying vertices with odd degree. How can we turn this result into something that talks about odd-degree vertices? Well: from our work in our first chapter, we know that every integer is either even or odd. If we apply this idea to our degree-sum formula, we get the following:

$$\boxed{\begin{array}{c}\text{The sum of the}\\ \textbf{odd} \text{ degrees of}\\ \text{vertices in } G\end{array}} + \boxed{\begin{array}{c}\text{The sum of the}\\ \textbf{even} \text{ degrees of}\\ \text{vertices in } G\end{array}} = \boxed{\begin{array}{c}\text{Two times the}\\ \text{number of the}\\ \text{edges in } G\end{array}}$$

We wanted to study the odd-degree vertices, so let's get them isolated on one side of our equation:

$$\boxed{\begin{array}{c}\text{The sum of the}\\ \textbf{odd}\text{ degrees of}\\ \text{vertices in } G\end{array}} = \boxed{\begin{array}{c}\text{Two times the}\\ \text{number of the}\\ \text{edges in } G\end{array}} - \boxed{\begin{array}{c}\text{The sum of the}\\ \textbf{even}\text{ degrees of}\\ \text{vertices in } G\end{array}}$$

On the right-hand side, notice that we have an even number (twice the number of edges) minus a bunch of even numbers (the degrees of all even-degree vertices in G); therefore, the right-hand-side is even!

As a result, the left-hand-side is also even. But this means that the sum of all odd-degree vertices is an even number.

We know that summing an odd number of odd numbers is always odd, and that summing an even number of odd numbers is always even. Because the left-hand side is even, we know we must be in the second case; that is, that we have an **even number of vertices of odd degree**, as claimed! $\square$

7.3 Proof by Cases

Our second proof technique is best illustrated by an example:

Theorem 7.2. *For every natural number n, if n is a **square** number , then $n \not\equiv 2$ mod 3.*

An integer n is said to be a **square** number if we can write $n = k^2$ for some other integer k. For example, $0, 1, 4, 9, 16, 25\ldots$ are all square numbers!

Proof. As always, we start by expanding our definitions. If n is a square number, then by definition we know that $n = k^2$ for some integer k.

From here, we use the particularly clever trick that this section is devoted to: we consider **cases**. That is: we want to look at what n is congruent to modulo 3.

We don't have any information about what n or k theirselves are modulo 3, so it would seem hard to introduce this information into our proof! However, by the definition of the modulus operator %, we know that every number is congruent to one of 0, 1 or 2 modulo 3. By definition, then, this means that we most always be in one of the following three cases:$k \equiv 0$ mod 3, $k \equiv 1$ mod 3 or $k \equiv 2$ mod 3.

In each of these cases, we can now expand our definitions and use our knowledge of modular arithmetic to proceed further:

1. Assume that we're in the $k \equiv 0$ mod 3 case. In this situation, we have that $k \equiv 3m$ for some m, which means that $k^2 = 9m^2 = 3(3m^2)$ is also a multiple of 3. Thus, $k^2 \equiv 0$ mod 3.
2. Now, assume instead that we're in the $k \equiv 1$ mod 3 case. In this situation, we have that $k \equiv 3m + 1$ for some m, which means that $k^2 = 9m^2 + 6m + 1 = 3(3m^2 + 2m) + 1$. Thus, $k^2 \equiv 1$ mod 3.
3. Finally, consider the last remaining case, where $k \equiv 2$ mod 3. In this situation, we have that $k \equiv 3m + 2$ for some m, which means that $k^2 = 9m^2 + 12m + 4 = 3(3m^2 + 4m + 1) + 1$. Thus, $k^2 \equiv 1$ mod 3.

In all three of these cases, we've seen that $n = k^2$ is not congruent to 2 modulo 3. These cases cover **all** of the possibilities! Therefore, we know that n is simply never congruent to 2 modulo 3 in any situation, and have therefore proven our claim. $\square$

The trick to the proof above was that we were able to introduce additional information about k (namely, its remainder on division by 3) by simply considering **all possible remainders** as separate cases! This technique, called **proof by cases**, is a powerful technique in several situations:

- Whenever you're dealing with integers and answering a question about whether some expression $f(n)$ is even or odd, try considering the two cases "n is even" and "n is odd."
- If you're dealing with a claim about modular arithmetic, or with claims like "is a multiple of", considering the different possible remainders that a number could have (i.e. the three cases where $k \% 3 = 0, 1$ or 2 we considered above) is often a great approach.
- If you're dealing with claims about rational and irrational numbers, separating the cases "x is rational" and "x is irrational" can be handy.

We practice this in the examples below:

Proof by cases is an *excellent* technique whenever you see an absolute value, as it lets you get rid of the absolute value in each case.

Claim 7.4. *For any real number x, we claim that $|x+7| - x \geq 7$.*

Proof. We proceed by considering cases:

- We first consider the case where $x \geq -7$. In this case, $x + 7 \geq 0$, and so $|x+7| - x$ is just $x + 7 - x = 7$. In this case, our inequality holds!
- Now, we consider the case where $x < -7$. In this case, $x + 7 < 0$, and so we have $|x+7| = -(x+7)$ (as the negative of a negative is a positive!)

  Therefore, we have $|x+7| - x = -(x+7) - x = -7 - 2x$. If $x < -7$¡ then $-2x > 14$, and so $-2x - 7 > 14 - 7 = 7$. Therefore our claim holds in this case as well!

Because every number is either greater than or equal to 7 or less than 7, we've considered all possible cases. As our claim was true in each possible case, this completes our proof! □

Claim 7.5. *For every two numbers x, y, we always have that* $\max(x,y) + \min(x,y) = x + y$.

Proof. We consider two possible cases:

- $x > y$. In this case, we have $\max(x,y) = x$ and $\min(x,y) = y$; therefore, $\max(x,y) + \min(x,y) = x + y$ as claimed.
- $x \leq y$. In this case, we have $\max(x,y) = y$ and $\min(x,y) = x$; therefore, $\max(x,y) + \min(x,y) = y + x = x + y$, also as claimed.

This covers all possible cases, as for any two numbers x, y either $x > y$ or $x \leq y$! Therefore, we've proven our claim. □

In the next example, we return to the tricks we used to calculate the last digit of a number in Claim 1.10:

Proof by cases is also usually a good idea if you see the modulus operator!

Claim 7.6. *For any integer k, we have that $(k^4) \% 10$ is always either 0, 1, 5, or 6.*

Proof. We saw before in our chapter on integers that if d_0 is the last digit of an integer n, then $n^m \% 10$ is equal to $d_0^m \% 10$ for any positive integer power m.

Therefore, in our claim, we don't have to actually consider every possible integer k; we can just consider the ten different possible last digits k could have, and calculate the cubes of each of those! We do so here:

- $0^4 \% 10 = 0 \% 10 = 0$.
- $1^4 \% 10 = 1 \% 10 = 1$.
- $2^4 \% 10 = 16 \% 10 = 6$.
- $3^4 \% 10 = 81 \% 10 = 1$.
- $4^4 \% 10 = 256 \% 10 = 6$.
- $5^4 \% 10 = 625 \% 10 = 5$.
- $9^4 \% 10 = (81)^2 \% 10 = 1^2 \% 10 = 1$.
- $6^4 \% 10 = (36)^2 \% 10 = 6^2 \% 10 = 6$.
- $7^4 \% 10 = (49)^2 \% 10 = 9^2 \% 10 = 6$.
- $8^4 \% 10 = (64)^2 \% 10 = 4^2 \% 10 = 6$.

In all ten cases, our remainders are always 0, 1, 5, or 6, as claimed! Therefore, we've proven our claim. □

Finally, we can use a proof by cases to prove one of our exercises:

Answer to Exercise 7.1. Even though there are lots of socks in the drawer, there are only 3 colours. Therefore, we can just take 4 socks to make sure that at least 2 of them are the same colour. To understand why, let's look at the colours of three first socks.

There are two possible cases here:

- If we were lucky enough to pick two matched socks from those first three, then we've succeeded!
- However, in the worst-case scenario the first three socks we took were all different colours, and we do not yet have a pair. In this situation, we have one sock of each colour.

  In this case, however, our fourth sock is guaranteed to match at one of our three previously chosen socks!

In any case, we've grabbed a pair of socks, as desired.

7.4 Proof by Contradiction

Contradiction — i.e. the "if we're stuck on a problem, suppose we're wrong and see what happens" proof technique — is a method we've already used to considerable success throughout this coursebook! In this section, we study several more examples of proof by contradiction, and talk a bit about the trickier aspects of this proof method.

To start, let's examine one of the most famous proofs by contradiction! In this proof, we're going to really pick apart the structure of a proof by contradiction, so that we can see *why* this method works:

Claim 7.7. *The number $\sqrt{2}$ is not rational.*

Proof. As always, let's start by unpacking our definitions:

- $\sqrt{2}$ is the unique positive real number such that when we square it, we get 2.
- A number x is **rational** if we can write $x = \frac{m}{n}$, where m and n are integers and n is nonzero.

With this done, our claim can be unpacked to the following:

"For a real number x, if $x = \sqrt{2}$, then there are no values of $m, n \in \mathbb{Z}$ with $n \neq 0$ such that $x = \frac{m}{n}$."

So: how do we do this? Because the problem wants us to show that we cannot write $\sqrt{2} = \frac{m}{n}$ for any integers m, n with $n \neq 0$, we can't just check a few examples: we'd have to look at *all* of them, and this could be quite difficult! We'd have to find some useful property that makes all examples of this form fail, and this could be quite hard to find.

Instead, consider the following way to "side-step" these difficulties. Instead of looking at all pairs m, n and trying to show that each one fails, let's assume that we *have* one such pair m, n such that $\sqrt{2} = \frac{m}{n}$!

With this assumption in hand, let's now show that this assumption "breaks mathematics" in some way: that starting from this assumption, we can get to something we know is impossible, like $1 + 1 = 0$. If we can do this, then we know that our original assumption that there was such a fraction $\frac{m}{n}$ must have been nonsense (i.e. false), and therefore that our claim that no such fraction exists is true!

We do this here. Suppose that we can find two integers m, n with $n \neq 0$ such that $\sqrt{2} = \frac{m}{n}$. If m and n have common factors, divide through by those factors to write $\frac{m}{n}$ in its simplest possible form: that is, don't write something like $\frac{3}{6}$ or $\frac{12}{24}$, write $\frac{1}{2}$

Then if we square both sides, we get $2 = \frac{m^2}{n^2}$. Multiplying both sides by n^2 gives us $2n^2 = m^2$, which means that m^2 is even (because it is a multiple of 2)!

This means that m is even (see the tutorials from earlier in this course!), and therefore that we can write $m = 2k$ for some integer k. If we plug this into our equation $2n^2 = m^2$, we get $2n^2 = (2k)^2 = 4k^2$, and by dividing by 2 we have $n^2 = 2k^2$.

This means that n^2 is even, and therefore that n is even as well (same logic as before.)

But this means that both n and m are multiples of 2; that is, that they have a common factor! We said earlier that we'd divided through by any common factors to get rid of them, so this is a **contradiction**: from our initial assumption we got to something that is both true and false. As a result, our original assumption (that we could write $\sqrt{2} = \frac{m}{n}$) must be false; that is, we have shown that $\sqrt{2} \neq \frac{m}{n}$ for any integers m, n with $n \neq 0$, as desired. Yay! □

We can generalize the form of the argument we just made above as follows:

- We have a claim we're trying to prove; let's denote it P, for shorthand.
- Instead of proving P is true directly, we want to prove that "not P" is impossible.
- To do this, we can simply do the following:
  1. Assume, for the moment, that "not-P" is actually true!
  2. Working from this assumption, find a pair of contradictory statements that are implied by "not P." That is, find a pair of statements Q and "not-Q" such that if P was false, both Q and "not-Q" would both hold. Common examples are "1=1" and "1=0", or "n is even" and "n is false", or "x is positive" and "x is negative": stuff like that.
  3. This proof demonstrates that "not-P" must be impossible, because it implies two contradictory things (like the two simultaneous claims "n is even"and "n is odd.") Mathematics is free from false statements and contradictions; therefore, we know that this must be impossible. In other words, "not-P" must be false and P must be true!

A beautiful quote about proofs by contradiction, by the mathematician G. H. Hardy: "[Proof by contradiction], which Euclid loved so much, is one of a mathematician's finest weapons. It is a far finer gambit than any chess gambit: a chess player may offer the sacrifice of a pawn or even a piece, but a mathematician offers the game."

This is how a **proof by contradiction** works. You take your claim P, assume it's false, and use "not P" to deduce contradictory statements, which you know mathematics cannot contain.

We consider another example here:

Claim 7.8. *There are two irrational numbers a and b such that a^b is rational.*

Proof. In the example we're studying here, we want to show that it's impossible for a^b to be irrational for every pair of irrational numbers a, b. To do this via a proof by contradiction, we do the following: first, assume that a^b **is** irrational for every pair of irrational numbers a, b! If we apply this knowledge to one of the few numbers $(\sqrt{2})$ we know is irrational, our assumption tells us that in specific

$$\sqrt{2}^{\sqrt{2}} \text{ is irrational.}$$

What do we do from here? Well: pretty much the only thing we have is our assumption, our knowledge that $\sqrt{2}$ is irrational, and our new belief that $\sqrt{2}^{\sqrt{2}}$ is **also** irrational. The only thing really left to do, then, is to let $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$, and apply our hypothesis again. But this is excellent! On one hand, our we have that a^b is irrational by our hypothesis. On the other hand, we have that a^b is equal to

$$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^{2} = 2,$$

which is clearly rational. This is a contradiction! Therefore, we know that our hypothesis must be false: there must be a pair of irrational numbers a, b such that a^b is rational. □

An interesting quirk of the above proof is that it didn't actually give us a pair of irrational numbers a, b such that a^b is rational! It simply told us that either

- $\sqrt{2}^{\sqrt{2}}$ is rational, in which case $a = b = \sqrt{2}$ is an example, or
- $\sqrt{2}^{\sqrt{2}}$ irrational, in which case $a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}$ is an example,

but it never actually tells us which pair satisfies our claim! This is a weird property of proofs by contradiction: they are often **nonconstructive** proofs, in that they will tell you that a statement is true or false without necessarily giving you an example that demonstrates the truth of that statement.

To stick with the classical route, let's study another one of the first proofs by induction, that we considered all the way back in our first chapter:

Claim 7.9. *(Euclid) There are infinitely many prime numbers.*

Proof. As we did with our argument that no number can be both even and odd at the same time, let's approach this with a bit of a thought experiment: what would happen if there were not infinitely many prime numbers?

Well: if this were to happen, then there would be some fixed number of primes in existence. Let's give that number a name, and say that there were n primes in existence. Then, if we had a piece of paper with n lines on it, we could in theory write down all of the prime numbers that existed!

If we labeled those lines $1, 2, \ldots n$, we could then refer to those prime numbers by their labels: that is, we could refer to our prime numbers by calling them $p_1, p_2, p_3, \ldots p_n$. (Giving things names: a very useful technique!)

In this world where we have all of these prime numbers, what can we do with them? Well: as we saw before, a particularly useful property about prime numbers is that they form the building blocks out of which we can make all integers. Therefore, we're motivated to take our primes and stick them together, and see what happens!

After a lot of effort, you might eventually hit on the clever combination of our prime numbers that Euclid discovered: think about what happens if we multiply all of our prime numbers together, and then add 1 to that entire sum. That is: look at the number

$$M = 1 + (p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_n)$$

On one hand: take any of the prime numbers on our list. To indicate that we're taking a general prime number from our list, let's refer to that prime number as p_i, where i could be any index. Look at $\frac{M}{p_i}$. By definition, this is equal to

$$\frac{1}{p_i} + \left(\overbrace{p_1 \cdot p_2 \cdot \ldots \cdot p_n}^{\text{all of the primes except for } p_i} \right).$$

In particular, notice that this is **not** an integer! $\frac{1}{p_i}$ is some fraction strictly between 0 and 1, because p_i is a prime and therefore at least 2, while the right-hand-bit is a product of integers and therefore is an integer itself.

Therefore, we've shown that if we multiply p_i by a number to get M, that number cannot be an integer; in other words, we have shown that p_i is not a factor of M. This holds for any of our primes, because p_i was an arbitrary prime; therefore M is not a multiple of **any** of our prime numbers!

On the other hand, though, we know that M is an integer. Therefore, we know that we can factor M into prime numbers! Do so, and write M as a product of prime numbers.

Our argument above tells us that that none of those prime numbers can be from our list $p_1, \ldots p_n$. But this list was supposed to contain **all** of the prime numbers! Therefore, we know that our original assumption that we could write down all of the prime numbers must have been false: that is, there must have been infinitely many prime numbers. □

7.5 Common Contradiction Mistakes: Not Understanding Negation

In a proof by contradiction, we're trying to prove that a claim P is true by showing that "not-P" cannot be false. Perhaps surprisingly, the most common mistake people make when using a proof by contradiction is in their very first step: specifically, in writing just what "not-P" is for a given claim!

For example, consider the following claim:

Claim 7.10. *For every pair of integers x, y such that x and y are both odd, we have that $x \cdot y$ is also odd.*

Here are a number of incorrect ways that people will try to negate this claim:

1. "For every pair of integers x, y such that x, y are both even, we have that $x \cdot y$ is also even."

   The first mistake made here is in the first two words, where we wrote "for every!" That is: Claim 7.10 is a claim about **all pairs** of integers. As such, if someone were to say that P was false, they'd just have to have one counterexample to prove us wrong!

   That is: if someone made a claim that every UoA student was enrolled in Compsci 120, you wouldn't prove them wrong by trying to show that every UoA student is not enrolled in Compsci 120; you'd just have to find **at least one** student not in Compsci 120.

   This tells us the first part of how we should write the negation of this claim: it should go "There is a pair of integers $x, y \ldots$"

2. This, however, is not enough! That is: "There is a pair of integers x, y such that x, y are both even and $x \cdot y$ is also even" is **also** not the negation of our claim.

   To see why this fails, note that Claim 7.10 is a claim about all pairs of **odd** integers. As such, if we're trying to say that this claim fails, we still need to work in the same universe as our normal claim, and we need to find a counterexample that consists of **a pair of odd integers**.

   That is: if someone told you "every Compsci 120 student who's never been to Antartica currently has an A," you don't disprove them by trying to find a student who's been to Antarctica! Their claim was about people who haven't been to Antarctica; to disprove it, you need to work within the same bounds!

As such, the **correct** negation of Claim 7.10 is the following:

"There is a pair of integers x, y such that both x, y are odd, and yet $x \cdot y$ is even."

Much more reasonable!

Let's consider another claim:

Claim 7.12. *If G is a graph containing ≥ 2 vertices, then G contains two vertices whose degrees are equal.*

There are many tempting and incorrect ways to negate this claim:

1. 'If G is a claim containing < 2 vertices, then G does not contain 2 vertices whose degrees are different." This is the same "negating the universe" error from before!

   That is: our claim is about graphs on 2 or more vertices. Its negation should still talk about graphs on 2 or more vertices! As well, our claim was about vertices whose degrees agreed. Its negation should still talk about vertices with equal degree!

2. This suggests the following as a fix: "If G is a graph containing ≥ 2 vertices, then G does **not** contains two vertices whose degrees are equal."

   This has the right universe, but it fails for a second, more subtle reason: the opposite of "if A then B" is **not** "if A then not-B."

   That is: suppose that someone claimed to you "if you attend tutorials in Compsci 120, you'll pass the class." The above strategy would say that you could disprove their claim by saying "if you attend tutorials in Compsci 120, then you won't pass the class."

   This doesn't really make sense, though! Their claim here is a really strong guarantee: it says that everyone who attends tutorials in Compsci 120 will pass. To disprove this, you don't need to show that **everyone** who attends tutorials won't pass; that's way too hard! Instead, you'd just need to find **at least one** person who (1) attended all of the tutorials but (2) didn't pass the class.

   That is: the opposite of "if A then B" is "there is a situation where A holds and B fails."

By using this, the correct negation of Claim 7.12 is the following:

"There is a graph G on $n \geq 2$ vertices, such that G does not contain two vertices with the same degree."

We can summarize the observations we made above as follows:

Using similar logic, the claim

Claim 7.11. *"There is an even prime number."*

should negate to the following:

"Every prime number is odd."

This is because the opposite of a claim about something existing is that there are no counterexamples (i.e. a claim about **everything**). As well, the universe of numbers we're studying (primes) should remain the same, leaving only the conclusion (even) to flip to "odd."

Observation 7.16. *The phrases "For every" and "There exist" get switched around when writing a proof by negation. This is because we disprove a claim about* ***everything*** *by finding a* ***single*** *counterexample, and we prove that* ***no example*** *of a thing can exist by showing that* ***everything*** *is not a counterexample!*

Observation 7.17. *The "universe" of a claim remains the same: i.e. we don't disprove a claim about all even numbers by studying odd numbers.*

Observation 7.18. *The opposite of an "if A then B" statement is "there is a situation where A holds and B fails." That is: if someone tells you that when it rains outside the sidewalk gets wet, you just need to find a situation where (1) it's raining and (2) some bit of sidewalk is still dry to disprove their claim!*

To finish this section and put this to use, let's prove Claim 7.12!

Proof of Claim 7.12. As noted above, the contradictive assumption here would be that G is a graph on $n \geq 2$ vertices in which all of the vertices have different degrees.

We know that the maximum degree of any vertex in G is $n-1$, because any vertex is at most adjacent to every **other** vertex. As well, the minimum degree of any vertex is trivially 0. Therefore, there are in theory n different possible degrees for the n vertices in G, namely the values $0, 1, \ldots n-1$.

If no degrees are repeated in G, then (because there are n vertices and n different possible degrees) there is exactly one vertex with degree i, for every $i \in \{0, 1, \ldots n-1\}$. If $n \geq 2$, note that in particular $n-1 \geq 1$, and so the degree-0 and degree-$(n-1)$ vertices are different.

Now, notice that if there is a vertex with degree $n-1$, it is connected to every other vertex in our graph. In particular it must be connected to the vertex that has degree 0, which contradicts the property that this vertex is supposed to have degree 0.

Therefore we have a contradiction, and can conclude that our original claim must hold. □

7.6 Proof by Construction

In many of the proofs above, we've been focused on proving claims about "all" numbers x, y, or "all" odd integers n, or other sorts of "universal" claims about things. When we're proving claims of these forms, we need to use techniques and arguments like the ones above where we work in general / don't get to use examples to prove our claim!

Sometimes, however, we'll find ourselves with claims of the form "There exists a number n such that..." or "There is a value x with the property..." In this sort of situation, we're not being asked to show that something is true for **all** values: instead, we're just asked to find a single example!

In situations like this, a common technique is **proof by construction**, where we simply create an object with the desired properties. We illustrate this with an example:

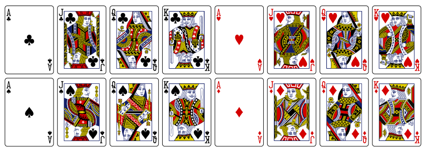
Claim 7.13. *There is an odd integer that is a power of two.*

Proof. Notice that $2^0 = 1$. Therefore, 1 is a power of 2. 1 is also odd, as we can write it in the form $1 + 2k$ for some integer k (specifically, $k = 0$.) Therefore we've constructed the claimed integer, as desired. □

Notice the following two aspects of this proof:

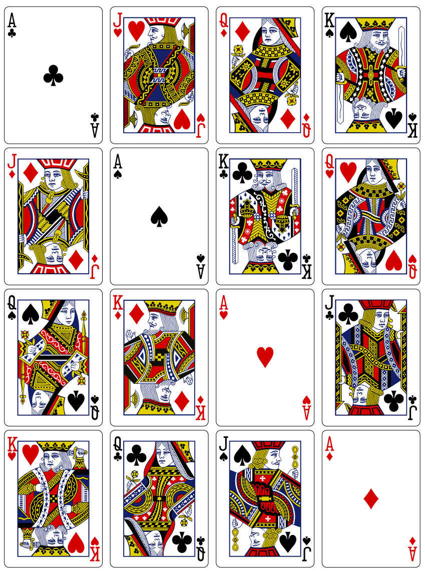
- We didn't have to work with a general integer n; instead, we got to give a specific example! This is because our claim was of the form "There is...", which means that we're just asked for a single example. If our proof had started "For all...", this would be different, and this proof would be invalid (just like how examples weren't enough for a proof in our earlier "the sum of any two odd numbers is even" claim.)
- Also notice that we didn't just say "1 is the answer" and ended our proof; we actually took the time to explain **why** 1 has the desired properties. You should expect to always do this!

We give a second example, to illustrate how these sorts of things come up in combinatorics and/or "puzzle" mathematics:



Claim 7.14. *Take the aces and face cards from a standard 52-card deck. Can you arrange them in a 4×4 grid so that no suits or symbols are repeated in any row or column?*

Proof. Behold!

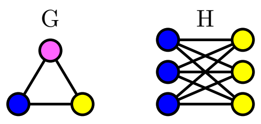


□

In this proof, we don't have much to really explain: the solution presented self-evidently has the desired property (just check every row and column.) If it was unclear, though, we'd have to have some explanation along with our answer!

We close by giving a pair of slightly trickier examples for how construction can work, by using **processes** and **algorithms**:

Definition 7.1. *Given a graph G, a vertex coloring of G with k colors is any way to assign each vertex of G one of k different colors, so that no two adjacent vertices get the same color.*



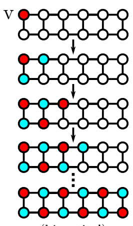
Claim 7.15. *We can vertex-color any tree T with at most 2 colors.*

Proof. Consider the following algorithm to paint any connected graph G's vertices with the colors red and blue:

Algorithm 7.13.

Init: Choose any vertex v in G, and paint it red.

1. *Take all currently uncolored vertices that are connected to any red vertices by an edge, and color them blue.*
2. *Take all currently uncolored vertices that are connected to any blue vertices by an edge, and color them red.*
3. *If there are any uncolored vertices left, go back to (i) and repeat.*



(bipartite!)

We claim that this algorithm will always succeed at coming up with a valid vertex coloring of any tree G; indeed, more generally, we claim that this algorithm will always succeed at making a valid 2-coloring of any graph G that doesn't contain an odd-length circuit as a subgraph! Because any tree does not have an odd-length circuit as a subgraph (indeed, it doesn't contain a cycle subgraph of any length), this would prove our claim.

To see why, we use a second proof technique: contradiction! Think about what would happen if this algorithm would fail, given a connected graph G with no odd-length circuit subgraphs.

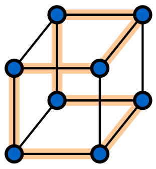
Because G is connected, the above process *will* eventually color every vertex of G; we first color v, then its neighbors, then its neighbor's neighbors, and so on/so forth, coloring every vertex within a walk of k edges by the k-th pass. So if the algorithm fails, it does so because in its coloring there must be an edge $\{x, y\}$ in which x, y both get the same color.

Notice that if a vertex w is colored **blue**, it is because we can walk to to w from our starting v in either one step, or three steps, or five steps . . . or in general an **odd** number of steps. This is because we alternated between red and blue in our algorithm. Similarly, if a vertex w is colored **red**, it is because we can walk to w in either 0 or 2 or 4 or 6 or . . . or an **even** number of edges.

Take any walk P_x from v to x, and any other walk P_y from v to y. We have proven that either P_x, P_y both have an even number of edges, or that they both have an odd number of edges. Therefore, the circuit formed by starting at v, walking along P_x to x, using the $\{x, y\}$ edge to go to y, and then reversing P_y to return to v has either (even + 1 + even) or (odd + 1 + odd) length. Both of the quantities, in particular, are **odd**! This contradicts our assumption that G had no odd-length circuits.

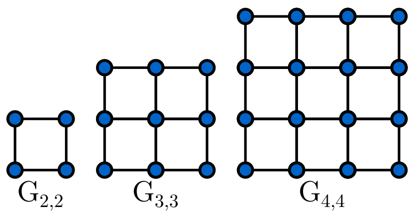
As a result, our original claim (that G is bipartite) must be true!

□



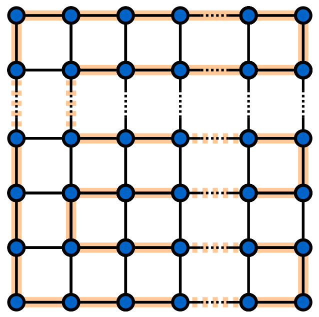
Claim 7.16. *A **Hamiltonian circuit** in a graph G is a walk that starts and ends at the same vertex, and along the way visits every other vertex exactly once. For example, the cube graph Q_3 drawn at right has a Hamiltonian circuit (highlighted.) has a Hamiltonian circuit (highlighted.)*

*The $n \times n$ **grid graphs** are defined by drawing a $n \times n$ grid of vertices and connecting adjacent vertices, as drawn below:*



Prove for all $n \in \mathbb{N}$ that $G_{2n,2n}$ has a Hamiltonian circuit.

Proof. Consider the following constructive process for generating such a circuit:



- Label the $(2n)^2$ vertices in the grid graph $G_{2n,2n}$ with coordinates (i,j), where vertex $(1,1)$ is the vertex in the bottom-left-hand corner and $(2n,2n)$ is the vertex in the upper-right-hand corner.
- Start at $(1,1)$,
- From this vertex, walk to the right until you're at the bottom-right corner $(1,2n)$.
- Go up one step to $(2,2n)$, and then walk back to the left until you're at $(2,2)$.
- Go up one step to $(3,2)$, then walk back to the right until you're at $(3,2n)$.
- Go up one step to $(4,2n)$, and then walk back to the left until you're at $(4,2)$.
- Go up one step to $(5,2)$, then walk back to the right until you're at $(5,2n)$.
- . . . Keep doing this! Eventually, you will find yourself at $(2n,2)$, having walked on all of the vertices whose second coordinates are not equal to 1, and not having visited any vertices whose second coordinate is 1 other than (1,1). (This is where the "$2n$" part comes in: because we go right on odd rows and left on even rows, if our grid has even height then we'll be going left on our top row and thus wind up at $(2n,2)$ as claimed.)
- Walk from $(2n,2)$ to $(2n,1)$, and then go down to $(1,1)$.

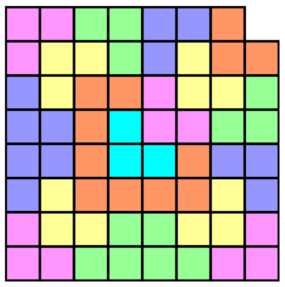
By construction we have visited all vertices in our graph exactly once, and thus created a Hamiltonian circuit, as desired. $\square$

Finally, we can answer one our earlier exercises by using construction:

Answer to Exercise 7.2. This has a nice constructive answer: take half of each of the four tablets today, and the other half tomorrow.

Because this is constructive, we don't have to explain how we came up with this clever idea: we can just present it as an answer! (Though if you did something clever to come up with an idea, it is good to mention how you did this.) We just have to explain why this works, which is pretty simple: half of each tablet gives us half of the tablets of each type, and thus exactly one dose for each kind.

7.7 Proof by Induction: First Examples



Sometimes, in mathematics, we will want to study a statement $P(n)$ that depends on some variable n. For example:

1. $P(n)$ = "The sum of the first n natural numbers is $\frac{n(n+1)}{2}$."
2. $P(n)$ = "If $q \geq 2$, we have $n \leq q^n$.
3. $P(n)$ = "Every polynomial of degree n has at most n roots."
4. $P(n)$ = Take a $2^n \times 2^n$ grid of unit squares, and remove one square from the top-right-hand corner of your grid. The resulting shape can be tiled by ▙ - shapes.

For any fixed n, we can usually use our earlier proof methods to prove that the claim holds! For instance, let $P(n)$ be the fourth example above, and consider $P(3)$, which is the claim that if we take a 8×8 grid of squares and delete the top-right-hand corner square, we can tile the rest of the shape with ▙ tiles. We can prove this by construction by just giving an explicit way to do it: see the drawing at right!

However, sometimes we will want to prove that one of these statements holds for **every** value $n \in \mathbb{N}$. How can we do this?

The answer here is **mathematical induction**! Mathematical induction is just a formal way of writing up our "building-block plus preserved property" process, in a way that will hopefully let us avoid everyone having the same shoe size. We describe it here:

- To start, take a claim $P(n)$ that we want to prove holds for every $n \in \mathbb{N}$.
- The first step in an inductive proof is the **base step**: in this step, we explicitly prove that the statement P holds for a few small cases using normal proof methods (typically construction or just calculation.)

  Usually you just prove that your claim holds when $n = 0$, but sometimes you start with $n = 1$ if your claim is one where 0 is a "dumb" case, or prove a handful of cases like $n = 0, 1, 2, 3$ to get the hang of things before moving on. You can think of this as the "building block" step from before!
- With this done, we move to the **induction step**! Here, we prove the following statement:

  If our claim P is true for all values up to some n,
  then it will **continue** to be true at the next value $n + 1$.

  Because this is an implication, i.e. an if-then proof, we usually prove it directly by assuming that our claim holds for all values up to some n, and then use this assumption to prove that our claim holds when we have $n + 1$ in our claim.

Just doing these two steps shows that your claim P is true for every natural number n! To see why, just examine what these two steps tell you:

- By our "base case" reasoning, we know that our claim is true at n=0.
- By our "inductive step" reasoning, we know that if our claim is true at 0, it is true at the "next" value $n + 1 = 1$.
- By our "inductive step" reasoning, we know that if our claim is true up to 1, it is true at the "next" value 2.
- By our "inductive step" reasoning, we know that if our claim is true up to 2, it is true at the "next" value 3.
- ...
- By continuing this process, we eventually get to any n! Therefore our claim is true for every $n \in \mathbb{N}$, as desired.

The way we usually think of inductive proofs is to think of toppling dominoes. Specifically, think of each of your $P(n)$ propositions as individual dominoes – one labeled $P(0)$, one labeled $P(1)$, one labeled $P(2)$, and so on/so forth. With our inductive step, we are insuring that all of our dominoes are *lined up* – in other words, that if we've knocked over some of them, the "next one" will also be knocked over. Then, we can think of the base step as "knocking over" the first domino. Once we do that, the inductive step makes it so that all of the later dominoes also have to fall, and therefore that our proposition must be true for all n (because all the dominoes fell!)

To illustrate how these kinds of proofs go, let's go back to our tiling problem, and prove that we can tile this grids for **every** $n \in \mathbb{N}$! (As an added bonus, let's prove it for grids where we remove one square from anywhere, not just the top-right-hand corner!)

Claim 7.17. *For any $n \in \mathbb{N}$, take a $2^n \times 2^n$ grid of unit squares, and remove one square from somewhere in your grid. The resulting grid can be tiled by - shapes.*

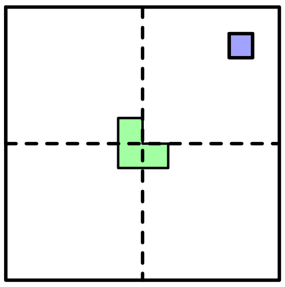
Proof. As suggested by the section title, we proceed by induction, where our proposition $P(n)$ is "we can tile a $2^n \times 2^n$ grid of 1×1 squares with one square deleted by using - shapes."

Base case: we want to prove $P(0)$. So: what *is* $P(0)$?

Well: for $n = 0$, we have a $2^0 \times 2^0 = 1 \times 1$ grid, which we've removed a 1×1 square from. In other words, we have **nothing**. If you want, you can think of "nothing" as being something we can trivially cover by placing no three-square shapes!

Alternately, you can decide that 0 is a stupid case and look at $n = 1$ instead. For $n = 1$, we simply have a 2×2 grid with one square punched out. As this *is* one of our three-square shapes, we are done here; just place a tile on top of our grid!

Either starting place is fine. In general, we recommend doing as many base cases as you need to do in order to feel comfortable with the pattern and believe that you've done something concrete! Most of the time, though, the base case will feel kinda silly; don't worry about this! The inductive step will do all of the heavy lifting for us.



Inductive step: We want to prove that if we know that our claim holds up to n, then it holds for $n+1$ as well; formally, this means that we want to show that if $P(0)$ and $P(1)$ and ... and $P(n)$ all hold, then $P(n+1)$ must follow.

In this problem in particular, this means that we're assuming that we can tile a $2^k \times 2^k$-grid with a square deleted for any $k \leq n$, and want to use this assumption to tile a $2^{n+1} \times 2^{n+1}$ grid with a square deleted.

To do this, take any $2^{n+1} \times 2^{n+1}$ grid with a square deleted. Divide it into four $2^n \times 2^n$ squares by cutting it in half horizontally and vertically. Finally, by rotating our grid if needed, make it so that the one missing square is in the upper-right hand corner.

Take this grid, and carefully cut out one three-square shape in the center as drawn at right.

Now, look at each of the four $2^n \times 2^n$ squares in this picture. They all are missing exactly one square: the upper-right hand one because of our original setup, and the other three because of our placed three-square-shape.

By our inductive hypothesis $P(n)$ we know that all of these smaller squares can be tiled! Doing so then gives us a tiling of the whole shape; in other words, we've shown how to use our $P(n)$ results to get a tiling of the $2^{n+1} \times 2^{n+1}$ grid.

As this completes our inductive step, we are thus done with our proof by induction. $\square$

The claim we proved above — one where we were some sense "growing" or "extending" a result on small values of n to get to larger values of n — is precisely the kind of question that induction is set up to solve! The Fibonacci numbers, which we introduce in the next question, is another object where this sort of "extension" approach is useful to consider.

Definition 7.2. *The* ***Fibonacci numbers*** f_n *are defined by a recurrence relation as follows:*

- $f_0 = 0$, $f_1 = 1$.
- *For any* $n \geq 2$, $f_n = f_{n-2} + f_{n-1}$.

To illustrate how it works, let's use it to calculate the first few values of the Fibonacci sequence! We know that $f_0 = 0, f_1 = 1$ by definition.

To find f_2, we can use the fact that for any $n \geq 2$, $f_n = f_{n-2} + f_{n-1}$ to calculate that.

$$f_2 = f_0 + f_1 = 0 + 1 = 1.$$

We can calculate further values of f_n similarly (see right!)

$$\begin{aligned} f_3 &= f_1 + f_2 = 1 + 1 = 2, \\ f_4 &= f_2 + f_3 = 1 + 2 = 3, \\ f_5 &= f_3 + f_4 = 2 + 3 = 5, \\ f_6 &= f_4 + f_5 = 3 + 5 = 8, \\ f_7 &= f_5 + f_6 = 5 + 8 = 13, \\ f_8 &= f_6 + f_7 = 8 + 13 = 21, \\ f_9 &= f_7 + f_8 = 13 + 21 = 34, \\ f_{10} &= f_8 + f_9 = 21 + 34 = 55, \\ f_{11} &= f_9 + f_{10} = 34 + 55 = 89, \\ f_{12} &= f_{10} + f_{11} = 55 + 89 = 144, \\ &\vdots \end{aligned}$$

When doing this, you'll likely notice a number of interesting properties about the Fibonacci sequence: see

https://en.wikipedia.org/wiki/Fibonacci_number

for a ton of weird/beautiful properties these numbers have! We prove one of these properties here:

Claim: For any $n \in \mathbb{N}$, the n-th Fibonacci number is even if and only if n is a multiple of 3.

Proof. Let $P(n)$ denote the claim "(the n-th Fibonacci number is even) $\Leftrightarrow$ (n is a multiple of 3)." We want to prove that $P(n)$ holds for all $n \in \mathbb{N}$, and proceed to prove this claim by induction.

Our **base cases** are pretty easy to check! We calculated the Fibonacci numbers from f_0 to f_{12} above, and we can see that the only ones that are even are f_0, f_3, f_6, f_9 and f_{12}; so we know that $P(0), P(3), P(6), P(9)$, and $P(12)$ all hold.

We now move to the **inductive step**: here, we want to prove $P(0)$ and $P(1)$ and $P(2)$ and ... and $P(n)$, when all combined together, imply $P(n+1)$. We start with what we're assuming, namely that all of $P(0), P(1), \ldots P(n)$ are all true: that is, we're assuming that the k-th Fibonacci number is even if and only if it is a multiple of 3, for every $k \in \{0, 1, \ldots n\}$.

We want to prove $P(n+1)$, i.e. that the $n+1$-th Fibonacci number is even if and only if it is a multiple of 3.

So: let's consider cases! There are two possible cases for the value $n+1$: either it is a multiple of 3, or it's not.

- If $n+1$ is a multiple of 3, we can write $n+1 = 3k$ for some $k \in \mathbb{Z}$. Notice that this means that $n = 3k-1$ and $n-1 = 3k-2$, and in particular that both of the values $n, n-1$ are not multiples of 3!

  As a result, our inductive assumption tells us that f_n, f_{n-1} are both not even, because they're not multiples of 3! But being not-even just means that these numbers are both odd. As a result, because $f_{n+1} = f_n + f_{n-1}$ =odd+ odd= even, we have shown that f_{n+1} is even in this case.

- If $n+1$ is not a multiple of 3, then $n+1$ either has remainder 1 or 2 when we divide it by 3; this is because any number has remainder 0, 1 or 2 when divided by 3. This means we can write $n+1 = 3k+1$ or $3k+2$, for some $k \in \mathbb{Z}$.

  As a result, we can see that of the two numbers $n, n-1$, exactly one of them is a multiple of 3; if $n+1 = 3k+1$ then $n, n-1 = 3k, 3k-1$, and if $n+1 = 3k+2$ then $n, n-1 = 3k+1, 3k$. As a result, our inductive hypothesis tells us that exactly one of f_n, f_{n-1} are odd, and the other is even.

  Therefore, because $f_{n+1} = f_n + f_{n-1}$ =(one odd number plus one even number)= odd, we have shown that f_{n+1} is odd in this case.

So, by using strong induction, we have proven that f_n is even if and only if it is a multiple of 3! ☐

7.8 Induction: Two Recurrence Relations

As we just saw in the section above, induction is a useful tool to study recurrence relations! In this section, we continue with this theme, and use induction to prove Claim 4.2 and Claim 4.6 from our algorithms chapter.

Claim 7.18 (Claim 4.2)**.** *For every positive integer n, we have* $\texttt{InsertionSortSteps}(n) = \dfrac{3n^2+9n-10}{2}$.

Proof. We proceed by induction. First, we can notice that the table of values we calculated earlier validates our claim for the first few values of n:

n	1	2	3	4	5	6	7
`InsertionSortSteps(n)`	1	10	22	37	55	76	100
$\frac{3n^2+9n-10}{2}$	1	10	22	37	55	76	100

This gives us our **base case**.

For the **inductive step**, we proceed as always: we assume that our claim holds up to some value n, and seek to prove it for $n+1$.

In particular, if our claim holds up to some value n, we have

$$\texttt{InsertionSortSteps}(n) = \frac{3n^2+9n-10}{2}.$$

As well, by Claim **??**, we know that

$$\texttt{InsertionSortSteps}(n+1) = 3(n+2) + \texttt{InsertionSortSteps}(n).$$

By combining these together, we get

$$\texttt{InsertionSortSteps}(n+1) = 3(n+2) + \frac{3n^2+9n-10}{2} = \frac{6(n+2)}{2} + \frac{3n^2+9n-10}{2} = \frac{3n^2+15n+2}{2}.$$

But we know that

$$\frac{3(n+1)^2+9(n+1)-10}{2} = \frac{3n^2+6n+3+9n+9-10}{2} = \frac{3n^2+15n+2}{2}$$

.

In other words, we've shown that our claim holds at $n+1$, and have thus proven our claim by induction! □

We can study `MergeSort` in the same way:

Claim 7.19 (Claim 4.6)**.** $\texttt{MergeSortSteps}(2^k) = k\cdot 2^{k+2} - 2^{k+1} - 1$, *for every natural number k.*

Proof. We again proceed by induction. Again, as before, our previously-calculated table of values suffices for a **base case**:

k	1	2	3	4	5	6	7
$\texttt{MergeSortSteps}(2^k)$	11	39	111	287	703	1663	3839
$k\cdot 2^{k+2} - 2^{k+1} - 1$	11	39	111	287	703	1663	3839

With this established, we turn to the **inductive step**. Here, we again assume that our claim holds up to some value k, and seek to prove it for $k+1$.

In particular, if our claim holds up to some value k, we have

$$\texttt{MergeSortSteps}(2^k) = k \cdot 2^{k+2} - 2^{k+1} - 1$$

As well, by Claim 4.5, we know that

$$\texttt{MergeSortSteps}(2^{k+1}) = 1 + 4 \cdot 2^{k+1} + 2 \cdot \texttt{MergeSortSteps}(2^k).$$

Again, by combining these together, we get

$$\begin{aligned}\texttt{MergeSortSteps}(2^{k+1}) &= 1 + 4 \cdot 2^{k+1} + 2(k \cdot 2^{k+2} - 2^{k+1} - 1) \\ &= 1 + 2^{k+3} + 2k2^{k+2} - 2 \cdot 2^{k+1} - 2 \\ &= k \cdot 2^{k+3} + 2^{k+3} - 2^{k+2} - 1,\end{aligned}$$

But we know that

$$(k+1) \cdot 2^{(k+1)+2} - 2^{(k+1)+1} - 1 = k \cdot 2^{k+3} + 2^{k+3} - 2^{k+2} - 1$$

.

In other words, we've shown that our claim holds at $k+1$, and have thus proven our claim by induction! □

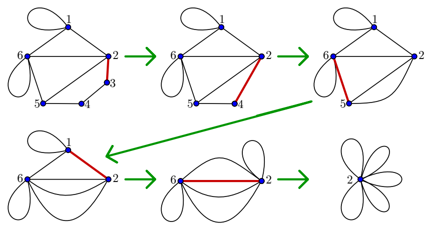
7.9 Induction, Graphs, and Trees

Induction is a particularly useful technique to use when studying graphs and trees! We prove three claims here, two of which you may recall from our section on trees:

Claim 7.20. *If G is a connected multigraph with loops (i.e. we allow multiple edges, and also allow an edge to have both of its endpoints be equal) on n vertices, then G contains at least $n-1$ edges.*

Proof. We proceed by induction on n. For $n = 0, 1$, this claim is trivially true, as we always have that E is a nonnegative number.

This establishes our base cases, so we now turn to the inductive step: here, we assume that our claim holds for all connected graphs on at most n vertices, and seek to use that assumption to prove that our claim holds for connected graphs on $n+1$ vertices.



To do this, consider the following operation, called **edge contraction**. We define this as follows: take any graph G and any edge e in G with two distinct endpoints. We define G_e, the graph that this edge, as follows: take G, delete e, and then combine e's two endpoints together into a single vertex, preserving all of the other edges that the graph has along the way.

We draw examples of this process at right: here, we have started with a graph on six vertices, and then contracted one by one the edges highlighted in red at each step.

Notice that contracting an edge decreases the number of vertices by 1 at each step, as it "squishes together" two adjacent vertices into one vertex. It also decreases the number of edges by 1 at each step, as we are contracting an edge to a point!

Finally, notice that contracting an edge preserves the property that our graph is connected. To see why, take any walk

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots \{v_{i-1}, v_i\}, \{v_i, v_{i+1}\}, \{v_{i+1}, v_{i+2}\}, \ldots \{v_{n-1}, v_n\}$$

in our graph. Notice that if we contracted an edge $\{v_i, v_{i+1}\}$ in this walk, this would collapse the vertices v_i, v_{i+1} into some new vertex $v_{i \oplus i+1}$ and preserve all of the edges other than $\{v_i, v_{i+1}\}$. As a result, our walk would just become

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots \{v_{i-1}, v_{i\oplus i+1}\}, \{v_{i\oplus i+1}, v_{i+2}\}, \ldots \{v_{n-1}, v_n\},$$

and thus still connects the vertices v_0, v_n. Therefore, edge contraction cannot "break" any pre-existing walks, and so preserves the property that our graph is connected.

We can use this process to prove our claim via induction:

- Take any connected multigraph graph G on $n+1$ vertices.
- Take any edge e in G with two distinct endpoints (such an edge exists, because G contains at least two different vertices and G is connected) and contract that edge. This gives us a new graph G_e, which is connected and contains n vertices.
- Therefore, by induction, we know that in G_e, the number of edges is at least $n-1$.
- We also know that G has exactly one more edge than G_e.
- Therefore, in G, we know that we have at least $n-1+1 = (n+1)-1$ edges. In other words, we've proven that our claim holds for graphs on $n+1$ vertices, as desired!

□

Notice that this result applies to simple graphs as well, as any simple graph is certainly a multigraph!

We can also use induction to prove Theorem 6.2! We split this result into two parts, as it's a longish equivalence proof:

Theorem 7.3 (Half of Theorem 6.2)**.** *If T is a tree on n vertices, then T contains exactly $n-1$ edges.*

Proof. We proceed by induction. Our base case is straightforward: any tree on 1 vertex clearly has no edges (as it's a simple graph.) If you want, you can also consider 2-vertex graphs as well; the only connected two-vertex graph is , which has one edge as desired.

For the inductive step, let's assume that our property holds for all trees on up to n vertices. Let T be any tree on $n+1$ vertices; we want to use our assumption to prove that T contains exactly $(n+1)-1 = n$ edges.

To do this, let l be a leaf vertex in T (we know that l exists by our earlier theorem.) Delete l and the edge connecting l to the rest of T from T; call the resulting graph $T-l$.

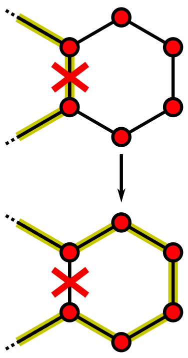
$T-l$ contains n vertices, because we started with $n+1$ vertices and deleted one vertex. It is also still connected (because l was degree 1, the only walk that would need to use the edge to l is a walk going directly to l, and we deleted l.) Finally $T-l$ contains no cycle subgraphs, because T contained no cycle subgraphs and deleting things from T cannot have somehow caused a cycle to exist.

Therefore $T-l$ is a tree! By induction, $T-l$ contains $n-1$ edges.

Therefore T itself contains $(n-1)+1 = n$ edges, because T is just $T-l$ plus the vertex l and the single edge connecting l to the rest of T. In other words, we've proven our inductive claim! □

Theorem 7.4 (The other half of Theorem 6.2)**.** *If G is a connected graph on n vertices containing exactly $n-1$ edges, then G is a tree.*

Proof. We proceed by contradiction; suppose that G is a connected graph on n vertices containing $n-1$ edges that is somehow not a tree.



Because G is connected, the only way that G can fail to be a tree is if it contains a cycle subgraph. Let $\{v_1, v_2\}, \{v_2, v_3\}, \ldots \{v_k, v_1\}$ be such a cycle subgraph.

Take G and delete the edge $\{v_1, v_2\}$ from G. We claim that G is still connected.

To see why, take any walk in G that uses the edge $\{v_1, v_2\}$, and replace each use of $\{v_1, v_2\}$ with the sequence of edges $\{v_1, v_k\}, \{v_k, v_{k-1}\}, \ldots \{v_3, v_2\}$. In other words, every time you'd go directly from v_1 to v_2 along that edge, instead use the cycle to go the "other" way around!

As a result, if two vertices x, y used to be connected by a walk in G, they are still connected after deleting $\{v_1, v_2\}$; in other words, $G - \{v_1, v_2\}$ is still connected.

But $G - \{v_1, v_2\}$ is a graph on n vertices containing $n - 2$ edges, as we had $n - 1$ edges and deleted one. But in Claim 7.20, we proved that a connected graph on n vertices must contain at least $n-1$ edges! In other words, we have a **contradiction,** and so our claim that G was a tree must have been correct. □

7.10 Proof Methods: How to Choose

With all of these proof methods at our fingertips, a natural question is this: how do you **choose** a method? One answer is the following:

Just try methods one-by-one until something works!

Paper is cheap, and it's usually just a lot faster to try stuff and see which things break than to predict ahead of time which method is "best." Also, most problems in maths can be solved by a number of different methods: there's rarely a single "correct" approach to a problem! Instead, many problems can be solved with many different techniques, and each different proof can help illustrate a new way of thinking about the task at hand.

With that said, though, there **are** clues or signs in a problem statement that can indicate that certain techniques might be useful. There are no hard-and-fast rules here, but the following observations often come in handy:

- Are you proving a claim of the form "if (some claim A is true), then (some other claim B is true)?" If so, a direct proof is maybe a good idea! Write down what it would mean for A to be true, and try to use that assumption to prove that B is also true.

- Are you dealing with modular arithmetic, even versus odd numbers, claims about "is a multiple of," or absolute values? Cases are often useful here. (More generally: if you have any problem where the inputs or outputs *can* be split into cases, do so! Proofs by cases often combine with other proof methods.)

- Are you being asked a claim of the form "Show that *blah* exists?" Construction's a good way to go here! (This is opposed to claims of the form "Show that every x has property foo," which you usually do not do by construction, as it's hard to construct *every* x!)

- Are you proving a claim where it seems like your previous results stick together to give you a later result? (Tiling problems that involve a general integer n, anything defined recursively like the Fibonacci sequence, processes that have recursion in them, ...) When you're writing your proof, do you find yourself wanting to use "..." to show how a pattern you've found continues? Then this is probably a good candidate for induction!

Induction is often especially useful for studying the runtime of an algorithm, or for proving that a given algorithm is "guaranteed" to give a specific output.

- Are you totally stuck? Try contradiction! Contradiction often gives you something to start from: i.e. it turns claims like "show that every object X has property *blah*" into "what would happen if an object X failed to have property *blah*?" This is often an easier place to start from! It can be a lot easier to think about how to "break" things and find contradictions of any kind, than to try to proceed directly and argue why some very specific property must hold.

  Contradiction is a particularly nice technique if you're trying to show that some task is impossible: the opening line of "suppose that this *is* possible" often makes proofs a lot easier to start.

- Are you *still* totally stuck? Maybe it's false: try a disproof! Best-case scenario: you disprove it and can move on. Worst-case scenario: even if you fail at disproving it, if you think about *why* you weren't able to disprove your claim, you might be able to turn that back into a good proof.

To get some practice with this, we solve a few problems below, and in each proof explain **why** we picked the methods that we did!

Claim 7.21. *For every positive integer n, $16^n - 1$ is a multiple of 15.*

Proof. Let's think about which of our proof methods we want to try:

- Direct proof: we could try this. This would involve expanding out what it means to be a multiple of 15, and trying to use logic/known results to get to the conclusion.
- Cases: even though cases is often a good technique when working with mods / multiple problems, this is likely not a great idea here. This is because there isn't really a clear set of cases you'd want to divide n into: even versus odd doesn't seem relevant, and considering all fifteen possible remainders of $n \% 15$ seems painful enough to not do unless absolutely necessary.
- Contradiction: could do, if we're stuck!
- Construction: not relevant. We're proving something for **every** integer, not building examples for **some** values.
- Induction: This doesn't obviously look like induction, in that it's not clear how you'd relate $16^n - 1$ to the "next" value $16^{n+1} - 1$.

  With some algebraic trickery, though, this is possible! Notice that $16(16^n - 1) = 16^{n+1} - 16 = (16^{n+1} - 1) - 15$, and thus that we've related one step to the next (in a way that involves a 15, which seems promising.) So if you saw this trick, then this is promising!

- Disproof: If you were suspicious of this claim, you could start by calculating a handful of values of $16^n - 1$, and see if any failed to be a multiple of 15.

  No obvious counterexamples immediately showed up in our table in the margins, so let's not try to disprove this just yet.

n	$16^n - 1$
0	1-1=0
1	$16 - 1 = 15$
2	$16^2 - 1 = 255 = 15 \cdot 17$
3	$16^3 - 1 = 4095 = 15 \cdot 273$
4	$16^4 - 1 = 65535 = 15 \cdot 4369$

So, amongst our proof methods, a **direct** proof and **induction** look promising. Let's try induction first!

Base case: we saw in our table in the margins that our case holds for $n = 0, 1, 2, 3$ and 4. So we've established our claim for a number of base cases.

Inductive step: For the inductive step, we assume that we've proven our claim for n: i.e. that $16^n - 1$ is a multiple of 15. We seek to use this claim to prove that our claim holds for the "next" value $n + 1$: i.e. that $16^{n+1} - 1$ is also a multiple of 15.

This is not too hard to do! Notice that as we observed above,

$$16^{n+1} - 1 = 16^{n+1} - 16 + 15 = 16(16^n - 1) + 15.$$

If $16^n - 1$ is a multiple of 15, then by definition we can write $16^n - 1 = 15k$ for some integer k. Doing so tells us that the right-hand-side is

$$16(15k) + 15 = 15(16k) + 15 = 15(16k + 1) = \text{a multiple of } 15.$$

Therefore, $16^{n+1} - 1$ is also a multiple of 15! As a result, we've proven our claim by induction: we showed that it holds for the first few values of n, and then showed that it will **stay** true, as if it is true for some value of n it must stay true for the "next" value $n + 1$. □

This is not the only way you could prove this result! We could also use a direct proof:

Proof. We want to show that $16^n - 1$ is always a multiple of 15, for any positive integer n.

By definition, this holds true if and only if $16^n \equiv 1 \bmod 15$.

So: we know that $16 \equiv 1 \bmod 15$, because $16 - 1$ is itself a multiple of 15. We also know from Claim **??** that for any positive integers a, b, c, n that if $a \equiv b \bmod c$, then $a^n \equiv b^n \bmod c$ as well.

Combining these facts tells us that for any positive integer n, we have $16^n \equiv 1^n \bmod 15$. Because $1^n = 1$ for all n, this gives us $16^n \equiv 1 \bmod 15$. By definition, this means that for every positive integer n we've shown that $16^n - 1$ is a multiple of 15, as desired! □

Claim 7.22. *Consider the following program* `puzzle(n)`*, which is a slightly modified version of the algorithm you studied in Practice Problem 5 on page 74. It takes in as input a nonnegative integer n, and does the following:*

```
(i)   If $n$ is either 0, 1, or 6, output $n$ and stop. Otherwise, go to (ii).
(ii)  If $n$ has two or more digits, replace $n$ with its last digit and go to (i). Otherwise, go to (iii).
(iii) Replace $n$ with $n^2$ and go to (i).
```

Prove that for every nonnegative odd number n, if `puzzle(n)` *stops, it outputs 1.*

Proof. We consider proof methods:

- Direct proof: we could try this, in that really any proof can be written in a direct method. (In this sense, "direct" often just means "not worrying about a specific technique.")
- Cases: This seems promising! Our program does different things based on different inputs. As such, cases is a natural technique to use!
- Contradiction: could do, if we're stuck!
- Construction: not relevant. We're proving something for **every** integer, not building examples for **some** values.

- Induction: Not a great technique here. It doesn't look like knowing what our program does on input n would tell us much about what it does on input $n+1$.
- Disproof: this seems true (run the program on a bunch of odd values if you're skeptical!), so disproving it doesn't seem like a good idea.

Cases looked like the strongest approach: so let's try that!

Take any nonnegative odd number n. Because n is a nonnegative integer, this means that $n = 1, 3, 5, 7, 9$ or n is a two-digit number (whose last digit is 1,3,5,7 or 9.)

After one iteration of our program, if n was a two-digit number it will be replaced with one of 1,3,5,7,9; so it suffices to just understand those five cases:

- If $n = 1$, then the program immediately stops and outputs 1, as desired.
- If $n = 3$, then on our first iteration we square n to get 9; on our second iteration we square again to get 81; on our third iteration we replace this two-digit number with 1; and on our fourth iteration we stop and output 1.
- If $n = 5$, we saw earlier that this case enters an infinite loop.
- If $n = 7$, then on our first iteration we square n and get 49; on our second iteration we replace this two-digit number with 9; on our third iteration we square to get 81; on our fourth iteration we replace this two-digit number with 1, which we then output and halt on our fifth iteration.
- If $n = 9$, we go to 81 and then 1 and then halt (as described above.)

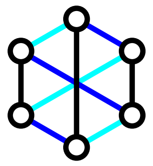
In all of these cases, we either run forever or output 1, as claimed! $\square$

Again, we can use a more direct approach if we see it:

Proof. Take any nonnegative odd number n. Notice that if n is odd, then no matter what our program does n will stay odd! This is because the square of an odd number is odd, and the last digit of an odd number is odd.

Therefore, n will never be reduced to either of 0 or 6. As a result, the only possibilities that remain are "the program halts when $n = 1$" or "the program runs forever," as there are no other conditions that cause our program to halt. $\square$

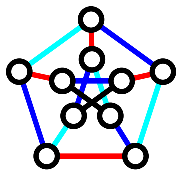
We close this section with a third problem about graphs:


a 3-edge-coloring

Claim 7.23. *Given a graph G, an edge coloring of G with k colors is any way to assign each edge of G one of k different colors, so that no two edges of the same color share an endpoint in common. An example of an edge coloring is given at right.*

*Show that there is a graph G in which all vertices have degree 3, and yet at least **four** colors are needed to create an edge-coloring of G.*

Proof. While we could go through all of the proof methods again, we'll shortcut the process and explain why we know this is a **constructive** proof: it's asking us to show that there is **a** graph with some property! This isn't a "show all graphs have property *foo*" problem or a "take any graph G, show that it cannot be *blah*" task: this is just asking us to find some single graph with a given property.

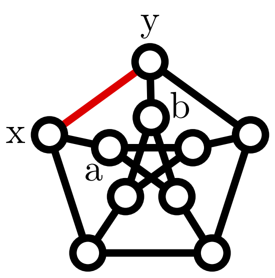


So, uh: behold the graph at right!

This is the Petersen graph P, a particularly useful counterexample to many claims in graph theory. We claim that P is a graph that needs four colors to properly color its edges; i.e. you cannot edge-color P with three colors. Note that to complete our proof, we need to explain why using just 3 colors to edge-color this grap is impossible: that is, it's not enough to just give our object, we also need to show that it has the desired property!

To do **this**, we now need a new proof technique. We claim that if you went through your list of proof techniques, none would stand out and you'd get stuck for a while. In this situation, **contradiction** is what we'd go to!

Here, a proof by contradiction would start as follows: suppose that we could use only three colors to color the edges of P. Call them red, green and blue.



Make the following observations:

1. Notice that because every vertex of P is degree 3, every vertex of P has one red, one green, and one blue edge leaving it.
2. Notice that on the outer pentagon of P, we need to use all three colors: if we tried to use just two colors to edge-color the pentagon, then we would have two edges with the same color touching each other.
3. Take a red edge on the outer pentagon of P. Call its two endpoints x, y, and let the inner vertices adjacent to x, y be called a, b. (Look at the picture in the margins to make sense of this!)

   Because $\{x, y\}$ is red, the edge $\{a, x\}$ is not red. Therefore, the red edge that (1) told us must be connected to a is on this inner star.

   Similarly, because $\{x, y\}$ is red, $\{b, y\}$ is not red. Therefore, the red edge that (1) told us must be connected to b is also on this inner star.

   Finally, because a and b are not adjacent, these red edges are different: that is, there are **two** red edges in this inner star.
4. Take a blue edge on the outer pentagon of P. The same logic as in (3) guarantees two blue edges in the inner star.
5. Take a green edge on the outer pentagon of P. The same logic as in (3) guarantees two green edges in the inner star.
6. Conclusion: the inner start has two blue edges, two red edges, and two green edges.

   ...but it only has **five edges!** Therefore this is impossible; i.e. we've reached a **contradiction**, and our original claim (that at least four colors are required) has been proven.

□

7.11 Practice Problems

1. You're a programmer! You've found yourself dealing with a program `mystery(n)` that has no comments in its code, and you want to know what it does. After some experimentation, you've found that `mystery(n)` takes in as input a natural number n, and does the following:

   (i) If n is either 0, 1, 2, or 3, output n and stop. Otherwise, go to (ii).

(ii) If n is even, replace n with $n/2$ and go back to (i). Otherwise, go to (iii).

(iii) Replace n with $n+5$ and go to (i).

Come up with the following proofs about `mystery(n)`:

(a) Use contradiction to prove the claim "if this program outputs 3 on input n, then n is not a power of 2."

(b) Disprove the claim "Given any natural number n as input, this program will eventually stop" by finding a counterexample.

(c) Prove by construction the claim "There is some input to this program that causes it to output 0."

(d) Write a direct proof that if "the output of this program is 1" then "the input to this program was 1."

2. (-) Let a, b, c be three integers, such that a divides b and a divides c. Write a direct proof that a also divides $b - c$.

3. (-) Write a proof by cases that for any integer n, the number $3n^2 + n - 16$ is even.

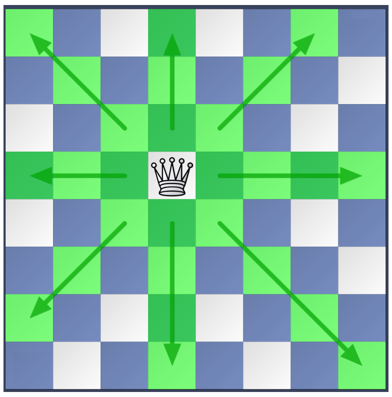
4. (-) Prove by contradiction that the number $\sqrt{19}$ is irrational.

5. Suppose that a, b are a pair of real numbers with the following property: if x is any number greater than b, then x must also be greater than a. Prove by contradiction that $a \leq b$.

6. (+) The game of **generalized n-tic-tac-toe** is played as follows: on a $n \times n$ grid, two players X and O take turns placing their respective symbols x, o into cells of the grid. No cell can be repeated. The game ends whenever any player gets n consecutive copies of their symbol on the same row /column / diagonal, or when the grid is completely filled in without any player having any such n consecutive symbols. (Normal tic-tac-toe is where $n = 3$.)

   Prove that there is no strategy in generalized tic-tac-toe where the **second player** to move is guaranteed to win.

7. A **queen** in the game of chess is a piece, shaped like ♕. In the game of chess, when moved, a queen (when placed in a given cell in a chessboard) can go to any cell within the same row, any cell within the same column, or any cell along the two diagonals through the cell that it starts from. We illustrate this in the margins.



   The **n-queens problem** is the following task: Take a $n \times n$ chessboard. Can you place n distinct queens on this chessboard, so that no queen can capture any other (i.e. so that there is no way to move any one queen into a cell currently occupied by another queen?)

   (a) Prove by cases that there are no solutions to the 3-queens problem.

   (b) Prove by construction that there is a solution to the 4-queens problem.

   (c) Prove by construction that there is a solution to the 8-queens problem.

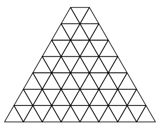
8. Prove or disprove the following claim: if G is a graph in which the degree of every vertex is 3, then G cannot be bipartite.

9. Prove or disprove the following claim: if G is a graph in which the degree of every vertex is at least 2, then G is connected.

10. Consider the following two-player game: starting with the single number 123, two players alternately subtract numbers from the set $\{1, 2, 3\}$ from this value. The player who first gets this sum to 0 wins.

    If you want to win this game, should you go first or second? Prove that your chosen player has a winning strategy. (Hint: try induction!)

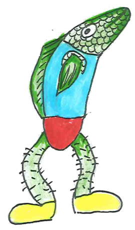
11. Take an equilateral triangle with side length 2^n. Divide it up into side-length 1 equilateral triangles, and delete the top triangle. Call this shape T_n:

    Take three side-length 1 equilateral triangles. Join them together to form the following tile: △▽△ Prove that you can tile T_n with △▽△ tiles, for every $n \in \mathbb{N}$.

12. Consider the following inductive "proof:"

    Claim: If G is a graph containing at least 3 vertices, and every vertex in G has degree at least 2, then G contains a C_3 subgraph

    .

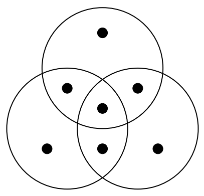
    Proof. We proceed by induction on the number of vertices in G. To start: we assume that our claim holds for all graphs on up to n vertices, and seek to prove that it holds for all graphs on $n + 1$ vertices as well.

    To do this: for any $n > 3$, take any graph G on n vertices in which every vertex has degree at least 2. Add a new vertex v to this graph, and connect v to at least two other vertices in G; this gives us a new graph G' on $n + 1$ vertices.

    We know by our inductive assumption that G itself contains a C_3 subgraph. Therefore this new graph G' on $n + 1$ vertices *also* contains a C_3 subgraph! This is what we wanted to prove, and thus finishes our inductive proof. $\square$

    Find **every** logical flaw in this proof. Explain why the flaws you have found are indeed mistakes. (Hint: there are at least two flaws here!)

13. Consider the following solitaire game:

    The picture above contains three circles drawn in the plane. In each of the bounded regions formed by the intersections of these circles, we've placed a coin, which is white on one side and black on the other. All of the coins start with their black side up.

The moves you're allowed to perform in this game are the following:

- You can at any time flip all of the coins within any circle.
- Alternately, you can at any time take any circle and flip all of its white coins over to black.

Can you ever reach the following configuration? Prove your claim.