David Liu & Toniann Pitassi

Mathematical Expression and Reasoning for Computer Science

Lecture Notes for CSC165 (Version 0.2)

Department of Computer Science
University of Toronto

Many thanks to Prof. Tom Fairgrieve for helpful comments and edits to an earlier version of these notes.

Contents

Prologue: what is this course about, and why should I care?

In this course, we will be talking about how to express statements precisely using the language of mathematical logic. This gives a way to communicate ideas without any ambiguity, which is an essential skill for any discipline. For example, the English statement "Some people like Trump" can be interpreted as saying that at least one person likes Trump, or that few, many, or even all people like Trump. What about "You can get cake or ice cream"? Does this mean that you may enjoy both cake and ice cream, or that you must choose between the two? Another example is the English expression "If you are a Pittsburgh Pens fan, then you are not a Philadelphia Flyers fan." Its meaning is clear enough if you meet a Pens fan, but what does this mean, if anything, for someone who *isn't* a Pittsburgh Pens fan? Does the same reasoning apply to the statement "If you can solve any problem in this course, then you will get an A"? Mathematical expressions in formal logic, on the other hand, have only one meaning. They remove all ambiguity so that only one interpretation is possible.

presumably Donald Trump likes Donald Trump

The second major theme of the course is developing methods to give rigorous mathematical proofs or disproofs of mathematical statements. We don't just want to be able to express ideas, we want to be able to argue – to both ourselves and others – that these ideas are correct. Mathematical proofs are a way to convince someone of something in an absolute sense, without worrying about biases, rhetoric, feelings, or alternate interpretations. The beauty of mathematics is that unlike other vast areas of human knowledge, it is possible to prove that a mathematical statement is true with one-hundred percent certainty. Without a rigorous mathematical proof, we can be easily fooled by our intuition and preconceptions. We will see throughout the course that some statements that seem perfectly reasonable turn out to be wrong, and others turn out to be true in surprising ways. Sometimes our intuition is valid and a proof seems like a mere formality; but often our intuition is in-

correct, and going through the process of a rigorous mathematic proof is the *only* way that we discover the truth!

Why do we need mathematical expression and reasoning in computer science? There are so many reasons! Perhaps the most basic one is program correctness. Say your friend has written a complicated program that she says does something truly remarkable. How do you know it is correct? You can test it on some inputs, but how do you know that your tests are thorough enough? A **correctness proof** will convince you that without a shadow of a doubt, the algorithm is correct on *all* possible inputs.

1. What does it *mean* for a program to "be correct?"
2. How can you *prove* that a program is correct?

But wait. Maybe her program does what she claims, but what if on some inputs it takes an extremely long time to run? A **worst-case complexity analysis** is a formal way to convince you that no matter what the input is, her program will run in some guaranteed number of time steps, independent of which computer or programming language is used to write and run this program.

1. What does it *mean* for a program to "take a long time to run?"
2. How can you *prove* that a program takes a long (or short) time to run?

These are two fundamental computer science areas where formal mathematical expression is required to precisely define concepts, and mathematical reasoning is required to prove statements about those concepts. Throughout this course we will follow this two-step process of **defining** and then **proving** things very explicitly, and we will practice on many examples. There are many other applications of mathematical expression and reasoning in computer science, some of which we list below. In all cases, mathematical expression allows us to precisely define our claims about the system in question, and mathematical proofs give us a mechanism to convince others with certainty that our system is working as we specified.

- *Program verification.* This is essentially program correctness mentioned above, and is in fact an entire subarea of computer science. Formal verification is the use of mathematical expression and reasoning in order to argue that a given software or hardware system is correct. Again, you need mathematical expression in order to specify without ambiguity both what the system is and what it means for the system to be correct. Then you need proofs in order to prove or disprove the correctness of the system.
- *Cryptography.* Cryptography is the science of developing techniques to communicate information in a way that is secure even in the presence of adversaries. The most basic cryptographic task is to send an *encrypted* message across the Internet to a particular person so that the intended receiver is able to decrypt the message, while ensuring that other agents, for whom the message is not intended, are not able

to modify the message or to decrypt it. The area of cryptography is now quite sophisticated, and there are extremely clever protocols that allow us to perform many tasks, such as public-key cryptography, digital signatures, and data authentication. Mathematical expression is required in order to even define precisely what we mean by "secure." Then proofs are needed in order to show that our cryptographic techniques are indeed secure.

You can think about it, but it is not at all obvious what such a definition should say. In fact, there are many definitions of security and other cryptographic notions used in theory and practice, depending on the context.

- *Privacy.* Issues of privacy are abundant. How do we manage the massive amount of data that is available through the web, while at the same time keep sensitive information private? In order to study this question, one first needs a formal definition of what is even meant by privacy. Intuitively, we want such a definition to capture the idea that data can be used for the benefit of society – such as to discover correlations between behaviour, symptoms and diseases – but so that the privacy of any particular person is not compromised. Once the definition is in place, the job then becomes to develop protocols and mechanisms that do useful things while maintaining a privacy guarantee. Again, one needs mathematical expression in order to state the definition of privacy, and proofs in order to show that the mechanisms satisfy the privacy definition.

As with "security", there are many definitions out there for what is meant by "privacy," including the notion of *differential privacy* that has lately been in the news.

- *Artificial Intelligence.* Many problems in artificial intelligence and machine learning involve logic. For example, in order to navigate a robot through a room, it helps to have a precise description of the room, as well as a plan for how to move through the room. Practically all problems in artificial intelligence involve mathematical expression and reasoning, including: natural language processing, image recognition, learning and planning.
- *Complexity Theory.* Complexity theory is about whether important problems that we want solve can be carried out efficiently with respect to costly resources. Common resources considered are time, computer memory, and randomness. This study requires formal definitions of what we mean by *efficient*; research in this area aims to invent proofs that certain problems can or cannot be solved efficiently.

The idea of "randomness" as a resource may be a surprising one, but is in fact the heart of one of the biggest open questions in complexity theory: If a problem can be solved by an efficient randomized algorithm, can it be solved by an efficient algorithm which has no randomness?

Course overview

In our first few weeks of this course, we will discuss mathematical expressions. That is, you will learn a new language and how to express precise statements in this language. It may seem daunting to pick up a new language in a few short weeks, but in fact you probably have been using this language since you were born. What we will do is formalize your intuitive understanding of logic so that it is as clear as possible

what constitutes a legal mathematical statement and what doesn't.

After learning how to express our statements in this language of mathematical logic, we will discuss ways of reasoning about the truth (or falsehood) of these statements. You will both read and write proofs, learning how to construct airtight arguments and communicate them to others, and how to poke holes in flawed proofs. To practice the dual skills of expression and reasoning in computer science domains, we will introduce several new domains to serve as the foundations for our mathematical statements: number theory, combinations and permutations, program runtime, and graphs.

Of course, we are not introducing these domains just for the sake of having a few new definitions to play around with. Each of the domains we will study in this course serve a vital role in many areas of computer science, which we will only scratch the surface of in this course.

1 Mathematical Expression

As a starting point for formalizing our intuition of logic, we will define two mathematical notions that we will use repeatedly throughout the course: sets and functions. Much of the terminology here may be review for you (or at least appear vaguely familiar), but please pay careful attention to the bolded terms, as we will make heavy use of each of them throughout the course. As we will stress again and again, *definitions* are precise statements about the meaning of a term or symbol; whenever we define something, it will be your responsibility to understand that definition so that you can understand (and later, reason about) statements using these terms at any point in the future.

Sets

Definition 1.1 (set, element, size, empty set)**.** A **set** is a collection of distinct objects, which we call **elements** of the set. A set can have a finite number of elements, or infinitely many elements. The **size** of a finite set A is the number of elements in the set, and is denoted by $|A|$. The **empty set** (the set consisting of zero elements) is denoted by $\emptyset$.

set

element

$|A|$

empty set

When students see new terminology for the first time, the definitions can feel abstract. As much as possible, we will strive in this course to pair new definitions with concrete examples to help you understand what these definitions actually mean.

Before moving on, let us see some concrete examples of sets. These examples illustrate not just the versatility of what sets can represent, but also illustrate various ways of *defining* sets.

Example 1.1. A finite set can be described by explicitly listing all its elements in curly brackets, such as $\{a, b, c, d\}$ or $\{2, 4, -10, 3000\}$.

Example 1.2. A set of records of all people that work for a small company. Each record contains the person's name, salary, and age. For example:

$$\{(\text{Ava Doe}, \$70000, 53), (\text{Donald Dunn}, \$67000, 30), (\text{Mary Smith}, \$65000, 25), (\text{John Monet}, \$70000, 40)\}.$$

Example 1.3. Here are some familiar *infinite* sets of numbers. Note that we use the ... to indicate the continuation of a pattern of numbers.

- The set of natural numbers, $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- The set of integers, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- The set of positive integers, $\mathbb{Z}^+ = \{1, 2, \ldots\}$.
- The set of rational numbers, $\mathbb{Q}$.
- The set of real numbers, $\mathbb{R}$.

By convention in computer science, 0 is a natural number..

Example 1.4. The set of all finite strings over $0, 1$. A *finite string over* $0, 1$ is a finite sequence $b_1 b_2 b_3 \ldots b_k$, where k is some positive integer (called its *length*) and each of b_1, b_2, etc. is either 0 or 1. The string of length 0 is called the *empty string*, and is typically denoted by the symbol ϵ.

For example, the length of the string 10100101 is 8.

Note that we have defined this set without explicitly listing all of its elements, but instead by describing exactly what properties its elements have. For example, using our definition, we can say that this set contains the element 01101000, but does not contain the element 012345.

Food for thought: how would you generate a list all finite strings over $0, 1$?

Example 1.5. A set can also be described as in this example:

$$\{x \mid x \in \mathbb{N} \text{ and } x \geq 5\}.$$

This is the set of all natural numbers which are greater than or equal to 5. The left part (before the $|$) gives a name to an element in the set, and right part states *conditions* on this element that must be satisfied to belong to the set.

This set definition can be read in English simply by replacing the $|$ with the word "where."

As a more complex example, we can define the set of rational numbers as follows:

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p, q \in \mathbb{Z} \text{ and } q \neq 0 \right\}$$

We have only scratched the surface of the kinds of objects we can represent using sets. Later on in the course, we will enrich our set of examples by studying sets of computer programs, sequences of numbers, and graphs.

Operations on sets

We have already seen one set operation already: the size operator, $|A|$. Here is a more comprehensive list of the common operations we can perform on sets that we will use in this course. The following operations return either True or False. We only describe when these operations return True; they return False in all other cases.

- $x \in A$: returns True when x is an element of A. $y \notin A$ returns True when y is *not* an element of A.

- $A \subseteq B$: returns True if every element of A is also in B. We say in this case that A is a **subset** of B. A set A is always a subset of itself: $A \subseteq A$ is always True.

subset

- $A = B$: returns True if $A \subseteq B$ and $B \subseteq A$. In this case, A and B contain the exact same elements.

The following operations return sets:

- $A \cup B$, the **union** of A and B. Returns the set consisting of all elements that occur in A, in B, or in both.

union

- $A \cap B$, the **intersection** of A and B. Returns the set consisting of all elements that occur in both A and B.

intersection

- $A \backslash B$, the **difference** of A and B. Returns the set consisting of all elements that are in A but that are not in B.

difference

- $A \times B$, the **(Cartesian) product** of A and B. Returns the set consisting of all *pairs* (a, b) where a is an element of A and b is an element of B.

product

- $\mathcal{P}(A)$, the **power set** of A, returns the set consisting of *all* subsets of A. For example, if $A = \{1, 2, 3\}$, then

Food for thought: what is the relationship between $|\mathcal{P}(A)|$ and $|A|$?

$$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Functions

Let A and B be sets. A **function** $f : A \to B$ is a mapping from elements in A to elements in B. A is called the *domain* of the function, and B is called the *range* of the function. For example, if A and B are both the set of integers, then the (predecessor) function $Pred : \mathbb{Z} \to \mathbb{Z}$, defined by $Pred(x) = x - 1$, is the function that maps each integer x to the next smallest integer.

domain, range

Functions can have more than one input. For sets $A_1, A_2, \ldots, A_k$ and B, a **k-ary function** $f : A_1 \times A_2 \times \cdots \times A_k \to B$ is a function that takes k arguments, where for each i between 1 and k, the i-th argument of f must be an element of A_i, and where f returns an element of B.

Some common English terms: unary, binary, and ternay functions take one, two, and three inputs, respectively.

For example, the addition operator $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a binary function that takes two real numbers and returns their sum. For readability, we usually write this function as $x + y$ instead of $+(x, y)$.

A **predicate** is a function whose range is {True,False }. For example, the predicate $Odd : \mathbb{N} \to \{\text{True}, \text{False}\}$ is a mapping from the natural numbers to {True, False }. All even numbers are mapped to False, and all odd numbers are mapped to True.

predicate

In more advanced studies you will often see False and True represented as 0 and 1, respectively.

Predicates and sets have a natural equivalence that we will sometimes make use of in this course. Given a predicate $P : A \to \{\text{True}, \text{False}\}$, we can define the set $\{x \mid x \in A \text{ and } P(x) = \text{True}\}$, i.e., the set of elements of A which satisfy P. We say an element $x \in A$ satisfies P when $P(x)$ is True. On the flip side, given a subset $S \subseteq A$, we can define the predicate $P : A \to \{\text{True}, \text{False}\}$ by $P(x) = \text{True}$ if $x \in S$, and $P(x) = \text{False}$ if $x \notin S$.

Summation and product notation

When performing calculations, we'll often end up writing sums of terms, where each term follows a pattern. For example:

$$\frac{1 \cdot 1^2}{3+1} + \frac{2 \cdot 2^2}{3+2} + \frac{3 \cdot 3^2}{3+3} + \cdots + \frac{100 \cdot 1^2}{3+100}$$

We will often use *summation notation* to express such sums concisely. We could rewrite the previous example simply as:

$$\sum_{i=1}^{100} \frac{i \cdot 1^2}{3+1}$$

In this example, i is called the *index of summation*, and 1 and 100 are the *lower* and *upper bounds* of the summation, respectively. A bit more generally, for any pair of integers j and k, and any function $f : \mathbb{Z} \to \mathbb{R}$, we can use summation notation in the following way:

$$\sum_{i=j}^{k} f(i) = f(j) + f(j+1) + f(j+2) + \cdots + f(k).$$

A special case: when $j > k$, we say that the summation is *empty*, and has a value of 0.

In other words, when $j > k$, $\sum_{i=j}^{k} f(i) = 0$.

We can similarly use *product notation* to abbreviate multiplication:

$$\prod_{i=j}^{k} f(i) = f(j) \times f(j+1) \times f(j+2) \times \cdots \times f(k).$$

When $j > k$, $\prod_{i=j}^{k} f(i) = 1$.

Fun fact: the Greek letter Σ (sigma) corresponds to the first letter of "sum", and the Greek letter Π (pi) corresponds to the first letter of "product."

Propositional Logic

We are now ready to begin our study of the formal language of logic. *Propositional logic* is a very elementary system of logic that involves only

propositions and simple logical connectives. On its own, it is too simple to formalize the things that we will want to talk about, but it is a crucial building block underlying other, more expressive systems of logic that we will need in this course.

Definition 1.2 (proposition, propositional variable)**.** A **proposition** is a statement that is either True or False. Examples of propositions are:

- $2 + 4 = 6$
- $3 - 5 > 0$
- Every even integer greater than 2 is the sum of two prime numbers.
- Python's implementation of `sort` works properly on every input list.

We use **propositional variables** to represent propositions; by convention, propositional variable names are lowercase letters starting at p.

The concept of a propositional variable is different from other forms of variables you have seen before, and even ones that we will see a bit later in this chapter. Here's a rule of thumb: if you see an expression involving a propositional variable p, you should be able to replace p with the statement "CSC165 is cool" and still have the expression make sense.

A **propositional formula** is a mathematical expression that is built up from propositional variables by applying certain logical operators, which are unary and binary predicates that take truth values and return truth values.They are described in the following subsections.

It is important to keep in mind when reading about these operators that they inform both the *structure* of formulas (what they look like) as well as the *truth value* of these formulas (what they mean: whether the formula is True or False based on the truth values of the individual propositional variables).

The basic operators NOT, AND, OR

The unary operator **NOT** is denoted by the symbol $\neg$. It negates the truth value of its input. So if p is True, then $\neg p$ is False, and vice versa. This is shown in the *truth table* at the side.

p	$\neg p$
False	True
True	False

Table 1.1: Truth table for **NOT** ($\neg$).

The binary operator **AND** is denoted by the symbol $\wedge$. It returns True if both its arguments are True, and otherwise it returns False.

p	q	$p \wedge q$
False	False	False
False	True	False
True	False	False
True	True	True

Table 1.2: Truth table for **AND** ($\wedge$).

The binary operator **OR** is denoted by the symbol $\vee$, and returns True if one or both of its arguments are True.

p	q	$p \vee q$
False	False	False
False	True	True
True	False	True
True	True	True

Table 1.3: Truth table for **OR** ($\vee$).

The truth tables for **AND** and **NOT** agree with the popular English usage of the terms; however, the operator **OR** may seem somewhat different from your intuition, because the word "or" has two different meanings to most English speakers. The English statement "You can have cake or ice cream." From a nutritionist, this might be an *exclusive or*: you can have cake or you can have ice cream, but not both. But from a kindly relative at a family reunion, this might be an *inclusive or*: you

can have both cake and ice cream if you want! The study of mathematical logic is meant to eliminate the ambiguity by picking one meaning of **OR** and sticking with it. In our case, we will always use **OR** to mean the *inclusive or*, as illustrated in the last row of its truth table.

What if we really wanted to say that you are allowed to have one or the other, but not both? This would be the **XOR**, or "exclusive or" operator, denoted by the symbol $\oplus$.

AND and **OR** are similar in that they are both *binary* operators on propositional variables. However, the distinction between **AND** and **OR** is very important. Consider for example a rental agreement that reads "first and last months' rent *and* a $1000 deposit" versus a rental agreement that reads "first and last months' rent *or* a $1000 deposit." The second contract is fulfilled with much less money down than the first contract.

The implication operator

One of the most subtle and powerful relationships between two propositions is *implication*, which is represented by the symbol $\Rightarrow$. The implication $p \Rightarrow q$ asserts that whenever p is True, An example of a logical implication in English is the statement: "If you push that button, then the fire alarm will go off." q must also be True. Implications are so important that the parts have been given names. The statement p is called the *hypothesis* of the implication and the statement q is called the *conclusion* of the implication.

In some contexts, we can think of logical implication as the temporal relationship that q is inevitable if p occurs.

How should the truth table be defined for $p \Rightarrow q$? First, when both p and q are True, then $p \Rightarrow q$ should be True, since when p occurs, q also occurs. Similarly, it is clear that when p is True and q is False, then $p \Rightarrow q$ is False (since then q is not inevitably True even when p is True). But what about the other two cases, when p is False and q is either True or False? This is another case where our intuition from both English language it a little unclear. Perhaps somewhat surprisingly, in both of these remaining cases, we will still define $p \Rightarrow q$ to be True.

p	q	$p \Rightarrow q$
False	False	True
False	True	True
True	False	False
True	True	True

Table 1.4: Truth table for **implication** ($\Rightarrow$).

vacuous truth

The two cases when $p \Rightarrow q$ is True but p is False are called the **vacuous truth** cases. How do we justify this assignment of truth values? The key intuition is that because the statement doesn't say anything about whether or not q should occur when p is False, it cannot be disproven when p is False. In our example above, if the alarm button is *not* pushed, then the statement is not saying anything about whether or not the fire alarm will go off. It is entirely consistent with this statement that if the button is not pushed, the fire alarm can still go off, or may not go off.

This explanation may be a little unsatisfying. Another very important reason for defining these vacuous truth cases is that it makes expressing and proving complex statements much more convenient than any other possible truth values.

The formula $p \Rightarrow q$ has two equivalent formulas which often prove useful. To make this concrete, we'll use our example "If you are a Pittsburgh Pens fan, then you are not a Flyers fan" from the introduction.

The following two formulas are equivalent to $p \Rightarrow q$:

Here, "equivalent" means that the two formulas have the same truth values; for any setting of their propositional variables to True and False, the formulas will either both be True or both be False.

- $\neg p \vee q$. On our example: "You are not a Pittsburgh Pens fan, or you are not a Flyers fan." This makes use of the vacuous truth cases of implication, in that if p is False then $p \Rightarrow q$ is True, and if p is True then q must be True as well.
- $\neg q \Rightarrow \neg p$. On our example: "If you *are* a Flyers fan, then you are *not* a Pittsburgh Pens fan." Intuitively, this says that if q doesn't occur, then p cannot have occurred either, because $p \Rightarrow q$ means that if p occurs, q must occur.

  This equivalent formula is in fact so common that we give it a special name: the **contrapositive** of the implication $p \Rightarrow q$.

contrapositive

There is one more related formula that we will discuss before moving on. If we take $p \Rightarrow q$ and switch the hypothesis and conclusion, we obtain the implication $q \Rightarrow p$, which is called the **converse** of the original implication.

converse

Unlike the two formulas in the list above, the converse of an implication is *not* logically equivalent to the original implication. Consider the statement "If you can solve any problem in this course, then you will get an A." Its converse is "If you will get an A, then you can solve any problem in this course." These two statements certainly don't mean the same thing!

Biconditional

The final logical operator that we will consider is the *biconditional*, denoted by $p \Leftrightarrow q$. This operator is True if the implication $p \Rightarrow q$ and its converse $q \Rightarrow p$ are both True.

p	q	$p \Leftrightarrow q$
False	False	True
False	True	False
True	False	False
True	True	True

Table 1.5: Truth table for **biconditional implication** ($\Leftrightarrow$).

In other words, $p \Leftrightarrow q$ is an abbreviation for $(p \Rightarrow q) \wedge (q \Rightarrow p)$. A nice way of thinking about the biconditional is that it asserts that its two arguments have the *same* truth value.

While we could use the natural translation of $\Rightarrow$ and $\wedge$ into English to also translate $\Leftrightarrow$, the result is a little clunky: $p \Leftrightarrow q$ becomes "if p then q, and if q then p." Instead, we often shorten this using a quite nice turn of phrase: "p if and only if q," which is often abbreviated to "p iff q."

Summary

We have now seen all five propositional operators that we will use in this course. Now is an excellent time to review these and make sure you

understand the notation, meaning, and English words used to indicate each one.

operator	notation	English
NOT	$\neg p$	"p is not true"
AND	$p \wedge q$	"p and q"
OR	$p \vee q$	"p or q (or both!)"
implication	$p \Rightarrow q$	"if p, then q"
bi-implication	$p \Leftrightarrow q$	"p if and only if q"

Exercise Break!

1.1 Decide if the following propositional formulas are tautologies. (A tautology is a formula that is True for every possible assignment of values to propositional variables).

(a) $((p \Rightarrow q) \wedge (p \Rightarrow r)) \Leftrightarrow (p \Rightarrow (q \wedge r))$.

(b) $(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$.

(c) $(\neg(p \vee q)) \Leftrightarrow (\neg p \wedge \neg q)$.

Predicate Logic

While propositional logic is a good starting point, most interesting statements in mathematics contain variables over domains larger than simply $\{\text{True}, \text{False}\}$. For example, the statement "x is a power of 2" is not a proposition because its truth value depends on the value of x. It is only after we *substitute* a value for x that we may determine whether the resulting statement is True or False. For example, if $x = 8$ then the statement becomes "8 is a power of 2", which is True. But if $x = 7$ then the statement becomes "7 is a power of 2", which is False.

A statement whose truth value depends on one or more variables from any set is a *predicate*: a function whose range is $\{\text{True}, \text{False}\}$. We typically use uppercase letters starting from P to represent predicates, differentiating them from propositional variables. For example, if $P(x)$ is defined to be the statement "x is a power of 2" then $P(8)$ is True since 8 is a power of 2, whereas $P(7)$ is False. Thus a predicate is like a proposition except that it contains one or more variables; when we plug in particular values for the variables, then the predicate becomes a proposition.

As with all functions, predicates can depend on more than one variable. For example, if we define the predicate $Q(x, y)$ to mean "$x^2 = y$," then $Q(5, 25)$ is True since $5^2 = 25$, but $Q(5, 24)$ is False.

Just as how common arithmetic operators like $+$ are really binary functions, the common comparison operations like $=$ and $<$ are *binary predicates*, taking two numbers and returning True or False.

We usually define a predicate by giving the statement that involves the variables, e.g. "$P(x)$ is the statement 'x is a power of 2."' However, there is another component which is crucial to the definition of a predicate: the *domain* that each of the predicate's variable(s) belong to. You must *always* give the domain of a predicate as part of its definition. So we would complete the definition of $P(x)$ as follows:

In higher-level courses the domain is often left implicit by the context of the predicate, but because this is an introductory course we are making everything as explicit as possible.

$$P(x) : \text{"}x \text{ is a power of 2," where } x \in \mathbb{N}.$$

Quantification of predicate variables

Unlike propositional formulas, a predicate by itself does not have a truth value: as we discussed earlier, "x is a power of 2" is neither True nor False, since we don't know the value of x! We have seen one way to obtain a truth value in substituting a concrete element of the predicate's domain for its input, e.g. setting $x = 8$ in the statement "x is a power of 2," which is now true.

However, we often don't care about a particular element, but rather some aggregation of the predicate's truth values over *all* elements of the domain. For example, the statement "the equation every real number x satisfies the inequality $x^2 - 2x + 1 \geq 0$" doesn't make a claim about a particular value of x, but rather *all possible* values of x!

Since there are infinitely many real numbers, this statement is actually asserting an infinite number of statements all at once.

There are two types of "truth value aggregation" we want to express; each type is represented by a *quantfier* that modifies a predicate by specifying how a certain variable should be interpreted.

The **existential quantifier** is written as $\exists$, and represents the concept of "*there exists* an element in the domain that satisfies the given predicate." For example, the statement

existential quantifier

$$\exists x \in \mathbb{N},\ x \geq 0$$

can be translated as "there exists a natural number x that is greater than or equal to zero." This statement is True since (for example) when $x = 1$, we know that $x \geq 0$. Note that there are many more natural numbers that are ≥ 0, which is perfectly fine. The existential quantfier says only that there has to be *at least one* element of the domain satisfying the predicate, but it doesn't say exactly how many elements do so.

One should think of $\exists x \in S$ as an abbreviation for a big **OR** that runs through all possible values for x from the domain S. We can expand the

quantified statement by substituting all possible natural numbers for x:

$$0 \geq 0 \vee 1 \geq 0 \vee 2 \geq 0 \vee 3 \geq 0 \vee \cdots$$

In this case the expression is infinite, because there are an infinite number of natural numbers.

The **universal quantifier** is written as $\forall$, and represents the concept that "*every* element in the domain satisfies the given predicate." For example, the statement

universal quantifier

$$\forall x \in \mathbb{N},\ x \geq 0$$

can be translated as "every natural number x is greater than or equal to zero." Again, this statement is True, since the smallest natural number is zero itself. However, the statement $\forall x \in \mathbb{N},\ x \geq 10$ is not True since not every natural number is greater than or equal to 10.

One should think of $\forall x \in S$ as an abbreviation for a big **AND** that runs through all possible values of x from S. Thus, $\forall x \in \mathbb{N},\ x \geq 0$ is the same as

$$0 \geq 0 \wedge 1 \geq 0 \wedge 2 \geq 0 \wedge 3 \geq 0 \wedge \cdots$$

Example 1.6. Let us look at a simple example of these quantifiers. Suppose we define $Loves(a, b)$ be a binary predicate that is True whenever person a loves person b. For example, the diagram on the right defines the relation "Loves" for two collections of people: $A = \{$Ella, Patrick, Malena, Breanna$\}$, and $B = \{$Laura, Stanley, Thelonious, and Sophia$\}$. A line between two people indicates that the person on the left loves the person on the right.

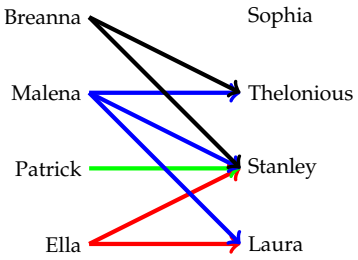


Figure 1.1: Sample *Loves* diagram.

We could have also justified this by saying that Breanna loves Thelonious instead.

Consider the following statements.

- $\exists a \in A, Loves(a, \text{Thelonious})$, which means "there exists someone in A who loves Thelonious." This is True since Malena loves Thelonious.
- $\exists a \in A, Loves(a, \text{Sophia})$, which means "there exists someone in A who loves Sophia." This is False since no one loves Sophia.
- $\forall a \in A, Loves(a, \text{Stanley})$, which means "every person in A loves Stanley." This is True.
- $\forall a \in A, Loves(a, Thelonious)$, which means "every person in A loves Thelonious." This is False.

Dealing with multiple quantifiers

It is usually straightforward to understand logical formulas with just a single quantifier, since they can generally be translated into English as either "there exists an element x of set S that satisfies $P(x)$" or "every element x of set S satisfies $P(x)$." However, we will often have situations

where there are multiple variables that are quantified, and we need to pay special attention to what such statements are actually saying.

For example, our *Loves* predicate is binary: what if we wanted to quantify *both* of its inputs? For example, consider the formula

$$\forall a \in A, \forall b \in B, Loves(a, b)$$

We translate this as "for every person a in A, for every person b in B, a loves b." After some thought, we notice that the order in which we quantified a and b doesn't matter; the statement "for every person b in B, for every person a in A, a loves b" means exactly the same thing! In both cases, we are considering all possible pairs of people (one from A and one from B).

So in general when we have two consecutive universal quantifiers the order does *not* matter. The following two formulas are equivalent:

- $\forall x \in S_1, \forall y \in S_2, P(x, y)$
- $\forall y \in S_2, \forall s \in S_1, P(x, y)$

Tip: when $S_1 = S_2$, we often abbreviate this to the more concise $\forall x, y \in S_1, P(x, y)$.

The same is true of two consecutive existential quantifiers. Consider the statements "there exist an a in A and b in B such that a loves b" and "there exist a b in B and a in A such that a loves b." Again, they mean the same thing: in this case, we only care about one particular pair of people (one from A and one from B), so the order in which we pick the particular a and b doesn't matter. In general, the following two formulas are equivalent:

- $\exists x \in S_1, \exists y \in S_2, P(x, y)$
- $\exists y \in S_2, \exists s \in S_1, P(x, y)$

So even though consecutive quantifiers of the same type behave very nicely, this is **not** the case for a pair of alternating quantifiers. First, consider

$$\forall a \in A, \exists b \in B, Loves(a, b).$$

This can be translated as "For every person a in A, there exists a person b in B, such that a loves b." Or put a bit more naturally, "For every person a in A, a loves someone in B."

This is true: every person in a loves at least one person.

a (from A)	b (a person in B who a loves)
Breanna	Thelonious
Malena	Laura
Patrick	Stanley
Ella	Stanley

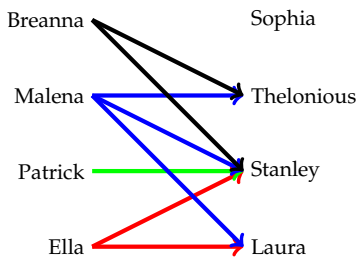


Figure 1.2: Sample *Loves* diagram.

Note that the choice of person who a loves depends on a: this is consistent with the latter part of the English translation, "a loves someone in B."

Let us contrast this with the similar-looking formula

$$\exists b \in B, \forall a \in A, Loves(a, b).$$

Even though this formula looks the same as the previous one (only the order of the quantifiers has changed), its English meaning is quite different: "there exists a person b in B, where for every person a in A, a loves b." Put more naturally, "there is a person b in B that is loved by everyone in A."

b (from B)	Loved by everyone in A?
Sophia	No
Thelonious	No
Stanley	Yes
Laura	No

This is True because all people in A loves Stanley. However, this would *not* be True if we removed the love connection between Malena and Stanley. In this case, Stanley would no longer be loved by everyone, and so *no one* in B is loved by everyone in A. But also notice that even if Malena no longer loves Stanley, the previous statement ("everyone in A loves someone") is still True!

So here we have a case where switching the order of quantifiers changes the meaning of a formula! In both cases, the existential quantifier $\exists b \in B$ involves making a *choice* of person from B. But in the first case, this quantifier occurs after a is quantified, so the choice of b is allowed to depend on the choice of a. In the second case, this quantifier occurs before a, and so the choice of b must be *independent* of the choice of a.

When reading a nested quantified expression, you should read it from left to right, and pay attention to the order of the quantifiers. In order to see if the statement is True, whenever you come across a universal quantifier, you must verify the statement for every single value that this variable can take on. Whenever you see an exists quantifier, you only need to exhibit *one* value for that variable such that the statement is True, and this value can depend on the variables to the *left* of it, but not on the variables to the right of it.

Sentences in predicate logic

Now that we have introduced the existential and universal quantifiers, we have a complete set of the tools needed to represent all statements we'll see in this course. A general formula in predicate logic is built up using the existential and universal quantifiers, the propositional operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$, and arbitrary predicates. To ensure that the formula has a fixed truth value, we want every variable in the formula to be quantified. We call a formula with no unquantified variables a **sentence**. So for example, the formula

Another common name for quantified variables is "bound variables," and unquantified variables are often called "free variables."

$$\forall x \in \mathbb{N},\ x^2 > y$$

is not a sentence: even though x is quantified, y is not, and so we cannot determine the truth value of this formula. We fix this by quantifying y as well, e.g.

$$\forall x, y \in \mathbb{N},\ x^2 > y.$$

Note that this expression is now a sentence, since both x and y are quantified. However, don't confuse a formula being a sentence with a formula being True! As we'll see repeatedly throughout the course, it is quite possible to express both True and False sentences, and part of our job will be to determine whether a given sentence is True or not, and to prove it.

Manipulating negation

We have already seen some equivalences among logical formulas, such as the equivalence of $p \Rightarrow q$ and $\neg p \vee q$. While there are many such equivalences, the only other major type that is important for this course are the ones used to simplify negated formulas. Taking the negation of a statement is extremely common, because often when we are trying to decide if a statement is True or not, it is useful to know exactly what its negation means, to decide whether that is more plausible than the original.

Given any formula, we can state its negation simply by wrapping it in a $\neg$ symbol:

$$\neg\big(\forall x \in \mathbb{N},\ \exists y \in \mathbb{N},\ x \geq 5 \vee x^2 - y \geq 30\big).$$

However, such a statement is rather hard to understand, if you try to transliterate each part separately: "Not for every natural number x,

there exists a natural number y, such that x is greater than or equal to 5 or $x^2 - y$ is greater than or equal to 30."

Instead, given a formula using negations, we apply some *simplification rules* to "push" the negation symbol to the right, closer the to individual predicates. Each simplification rule "moves the negation inside" by one step, giving a pair of equivalent formulas, one with the negation applied to one of the logical operator or quantifiers, and one where the negation is applied to the inner subexpression.

- $\neg(\neg p)$ becomes p.
- $\neg(p \vee q)$ becomes $(\neg p) \wedge (\neg q)$.
- $\neg(p \wedge q)$ becomes $(\neg p) \vee (\neg q)$.
- $\neg(p \Rightarrow q)$ becomes $p \wedge (\neg q)$.
- $\neg(p \Leftrightarrow q)$ becomes $(p \wedge (\neg q)) \vee ((\neg p) \wedge q))$.
- $\neg(\exists x \in S,\ P(x))$ becomes $\forall x \in S,\ \neg P(x)$.
- $\neg(\forall x \in S,\ P(x))$ becomes $\exists x \in S,\ \neg P(x)$.

The rules for AND and OR are also known as *deMorgan's laws*.

Since $p \Rightarrow q$ is equivalent to $\neg p \vee q$.

It is usually easy to remember the simplification rules for $\wedge$, $\vee$, $\forall$, and $\exists$, since you simply "flip" them when moving the negation inside. The intuition for the negation of $p \Rightarrow q$ is that there is only one case where this is False: when p has occurred but q does not. The intuition for the negation of $p \Leftrightarrow q$ is to remember that $\Leftrightarrow$ can be replaced with "have the same truth value," so the negation is "have different truth values."

What about the quantifiers? Consider a statement of the form $\neg(\exists x \in S,\ P(x))$, which says "there does not exist an element x of S that satisfies P." The only way this could be true is for every element of S to *not* satisfy P: "every element x of S does not satisfy P." A similar line of reasoning applies to $\neg(\forall x \in S,\ P(x))$.

Defining predicates

Throughout this course, we will study various mathematical objects that play key roles in computer science. As these objects become more complex, so too will our statements about them, to the point where if we try to write out everything using just basic set and arithmetic operations, our formulas won't fit on a single line! To avoid this problem, we create *definitions*, which we can use to express a long idea using a single term.

This is completely analogous to using temporary variables or helper functions in programming to express *part* of an overall value or computation.

In this section, we'll look at one extended example of defining our own predicates and using them in our statements. Let's take some terminology that is already familiar to us, and make it precise using the language of predicate logic.

Definition 1.3 (divisibility)**.** Let $n, d \in \mathbb{Z}$. We say that d **divides** n, or n **is divisible by** d, if and only if there exists a $k \in \mathbb{Z}$ such that $n = dk$.

You may be used to defining divisibility over just the natural numbers, but it will be helpful to allow for negative numbers in our work.

In this case, we use the notation $d \mid n$ to represent "d divides n." Note that just like the equals sign $=$ is a binary predicate, so too is $|$. For example, the statement $3 \mid 6$ is True, while the statement $4 \mid 10$ is False.

Example 1.7. Let's express the statement "For every integer x, if x divides 10, then it also divides 100" in two ways: with the divisibility predicate $d \mid n$, and without it.

- **With the predicate**: this is a universal quantification over all possible integers, and contains a logical implication. So we can write

$$\forall x \in \mathbb{Z},\ x \mid 10 \Rightarrow x \mid 100.$$

- **Without the predicate**: the same structure is there, except we *unpack the definition* of divisibility, replacing every instance of $d \mid n$ with $\exists k \in \mathbb{Z},\ n = dk$.

$$\forall x \in \mathbb{Z},\ \big(\exists k \in \mathbb{Z},\ 10 = kx\big) \Rightarrow \big(\exists k \in \mathbb{Z},\ 100 = kx\big).$$

  Note that each subformula in the parentheses has its own k variable, whose scope is limited by the parentheses. However, even though this technically correct, it's often confusing for beginners. So instead, we'll tweak the variable names to emphasize their distinctness:

That is, the k in the hypothesis of the implication is different from the k in the conclusion.

$$\forall x \in \mathbb{Z},\ \big(\exists k_1 \in \mathbb{Z},\ 10 = k_1 x\big) \Rightarrow \big(\exists k_2 \in \mathbb{Z},\ 100 = k_2 x\big).$$

As you can see, using this new predicate makes our formula quite a bit more concise! But the usefulness of our definitions doesn't stop here: we can, of course, use our terms and predicates in further definitions.

Definition 1.4 (prime)**.** A natural number p is **prime** if and only if it is greater than 1 and the only natural numbers that divide it are 1 and itself.

Keep in mind that unlike divisibility, here we restrict primes to being positive.

Example 1.8. Let's define a predicate $Prime(p)$ to express the statement that "p is a prime number", with and without using the definition of divisibility.

The first part of the definition, "greater than 1," is straightforward. The second part is a bit trickier, but a good insight is that we can enforce constraints on values through implication: if a number d divides p, then $d = 1$ or $d = p$. We can put these two ideas together to create a predicate formula:

$$Prime(p) : p > 1 \wedge \big(\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p\big),\ \text{where } p \in \mathbb{N}.$$

To express this idea without using divisibility, we need to unpack the $d \mid p$ expression, using the definition of divisibility. (The underline shows what's changed.)

$$Prime(p) : p > 1 \wedge \big(\forall d \in \mathbb{N},\ \underline{(\exists k \in \mathbb{Z},\ p = kd)} \Rightarrow d = 1 \vee d = p\big).$$

Example 1.9. Finally, let us express one of the more famous properties about prime numbers: "there are infinitely many primes."

Later on in this course, we'll actually prove this!

We have just seen how to express the fact that a single number p is a prime number, but how do we capture "infinitely many"? The key idea is that because primes are natural numbers, if there are infinitely many of them, then they have to keep growing bigger and bigger. So we can express the original statement as "every natural number has a prime number larger than it," or in the symbolic notation:

Another way to think about this is to consider the statement "every prime number is less than 9000." If this statement were True, then there could only be at most 8999 primes.

$$\forall n \in \mathbb{N},\ \exists p \in \mathbb{N},\ p > n \wedge Prime(p).$$

Of course, if we wanted to express this statement without either the *Prime* or divisibility predicates, we would end up with an extremely cumbersome statement:

$$\forall n \in \mathbb{N},\ \exists p \in \mathbb{N} p > n \wedge p > 1 \wedge \Big(\forall d \in \mathbb{N},\ (\exists k \in \mathbb{Z},\ p = kd) \Rightarrow d = 1 \vee d = p\Big).$$

This statement is terribly ugly, which is why we define our own predicates! Keep this in mind throughout the course: when you are given a statement to express, make sure you are aware of all of the relevant definitions, and make use of them to simplify your expression.

One last example: Fermat's Last Theorem

As payoff for the work that we have done so far, let us use predicate logic to express one of the most famous statements in mathematics: Fermat's Last Theorem. It was first conjectured by the mathematician Pierre de Fermat in 1637 in the margin of a copy of the text *Arithmetica*, where he claimed that he had a proof that was too large to fit in the margin! Despite this purported proof, for centuries this statement had no published proof. It wasn't until 1994 that Andrew Wiles finally proved this theorem.

Fermat's Last Theorem states that there are no three positive integers a, b, c that satisfy $a^n + b^n = c^n$ for any integer $n > 2$. To express this in predicate logic, we identify the relevant variables: a, b, c, and n. Are they universally or existentially quantified? The n certainly is universally quantified, since we say that the statement is True "for any $n > 2$." The statement also makes a claim that *no* a, b, c satisfy the given equation, which we can rephrase as "there do not exist a, b, c satisfying..." Finally, recall that we can express the condition $n > 2$ using an implication: *if* $n > 2$, *then* there is no solution to... Putting this together yields:

$$\forall n \in \mathbb{N},\ n > 2 \Rightarrow \neg\big(\exists a, b, c \in \mathbb{Z}^+, a^n + b^n = c^n\big).$$

We can now simplify this statement by pushing the negation inwards, so that this statement becomes

$$\forall n \in \mathbb{N},\ n > 2 \Rightarrow \big(\forall a, b, c \in \mathbb{Z}^+, a^n + b^n \neq c^n\big).$$

Our conventions for writing formulas

Mathematical expressions in predicate logic can become complicated very quickly. In order to avoid confusion and to make things as clear as possible we will follow some important conventions.

Operator precedence

The longer and more complex our formulas, the harder they are to read and understand. For example, here is a rather more complicated formula:

$$\forall x, y \in \mathbb{N},\ \exists z \in \mathbb{N},\ x + y = z \wedge x \cdot y = z \Rightarrow x = y.$$

Whenever we mix different propositional operators together, or when we mix quantifiers with formulas containing predicates, we need to worry about which ones come first – i.e., which ones have higher precedence. Techincally, we can just use parentheses around every operation, but this becomes very tiring very quickly. Instead, we will use the following precedence levels, in *decreasing* order of precedence.

1. $\neg$
2. $\vee$, $\wedge$
3. $\Rightarrow$, $\Leftrightarrow$
4. $\forall$, $\exists$

Combinations of operations at the same level *must* be disambiguated using parentheses.

So for example the expression

$$(p \vee \neg q) \wedge r \Rightarrow \big((s \vee t) \wedge u\big) \vee (\neg v \wedge w)$$

represents

$$\Big((p \vee (\neg q)) \wedge r\Big) \Rightarrow \Big(\big((s \vee t) \wedge u\big) \vee \big((\neg v) \wedge w\big)\Big),$$

and the expression

$$\forall x, y \in \mathbb{N},\ \exists z \in \mathbb{N},\ x + y = z \wedge x \cdot y = z \Rightarrow x = y$$

represents

$$\forall x, y \in \mathbb{N},\ \Big(\exists z \in \mathbb{N},\ \Big(\big(x + y = z \wedge x \cdot y = z\big) \Rightarrow x = y\Big)$$

Variable scope and naming

As we saw in the previous section, formulas involving multiple variables can be hard to keep track of. We will always use *distinct names for each variable* to ensure there is no possibility of confusion about what a variable is referring to. Here is an example, where f is a unary function from $\mathbb{N}$ to $\mathbb{N}$:

$$\big(\forall x \in \mathbb{N},\ f(x) \geq 5\big) \vee \big(\exists x \in \mathbb{N},\ f(x) < 5\big).$$

In this statement, we have two different occurrences of quantified variables, but they have the same name. We will always prefer to write it in this equivalent form, where each occurrence has a distinct name:

$$\big(\forall x \in \mathbb{N},\ f(x) \geq 5\big) \vee \big(\exists y \in \mathbb{N},\ f(y) < 5\big).$$

We do this even when expanding the same definition multiple times, typically using subscripts to differentiate the occurrences:

$$x \mid 10 \Rightarrow x \mid 100$$

becomes

$$\Big(\exists k_1 \in \mathbb{Z},\ 10 = k_1 x\Big) \Rightarrow \Big(\exists k_2 \in \mathbb{Z},\ 100 = k_2 x\Big).$$

Each quantification of a variable will be followed by a formula, which will be the *scope* of this variable. For example $\forall x \in \mathbb{N},\ f(x) \geq 5$ – the formula $f(x) \geq 5$ is the part of the statement that involves x.

Quantifiers are read left-to-right, which is why in $\forall a \in A, \exists b \in B$ the variable a is in scope when choosing b, but this is not true in $\exists b \in B, \forall a \in A$.

Finally, because we take quantifiers to have lowest precedence, the scope of a variable usually lasts until the end of the formula. The only time this is *not* the case is if the quantification is surrounded by parentheses, as in

$$\big(\forall x \in \mathbb{N},\ \underline{f(x) \geq 5}\big) \vee \big(\exists y \in \mathbb{N},\ \underline{f(y) < 5}\big).$$

Here, the scope of x is only the first underlined formula, and the scope of y is only the second underlined formula.

Exercise Break!

1.2 Let S be a set of people, C be the set of all countries, and let T be a predicate defined over $S \times C$ such that $T(x, y)$ is True if $x \in S$ has travelled to country $y \in C$. Express each of the following statements by a simple English sentence.

- $(\exists x \in S,\ T(x, \text{France})) \wedge (\forall y \in S,\ T(y, \text{Japan}))$.
- $\forall x \in S,\ \exists y \in C,\ T(x, y)$.
- $\forall x, z \in S,\ \exists y \in C,\ T(x, y) \Leftrightarrow T(z, y)$.

1.3 Write each of the statements below in logical form, and then give the contrapositive and converse of each statement.

- If all birds fly, and if Tweety is a bird, then Tweety flies.
- If it does not rain or it is not foggy, then the sailing race will be held and registration will go on.
- If rye bread is for sale at Ace Bakery, then rye bread was baked that day.

---

2 *Introduction to Proofs*

In the previous chapter, we studied how to express statements precisely using the language of predicate logic. But just as English enables to make both true and false claims, the language of predicate logic allows for the expression of both true and false sentences. In this chapter, we will turn our attention to analysing and communicating the truth (or falsehood) of these statements. You will develop the skills required to answer the following questions:

- How can you figure out if a given statement is true or false?
- If you know a statement is true, how can you convince others that it is true? How can you do the same if you know the statement is false instead?
- If someone gives you an explanation of why a statement is true, how do you know whether to believe them or not?

These questions draw a distinction between the internal and external components of mathematical reasoning. When given a new statement, you'll first need to figure out for yourself whether it is true (internal), and then be able to express your thought process to others (external). But even though we make a separation, these two processes are certainly connected: it is only after convincing yourself that a statement is true that you should then try to convince others. And often in the process of formalizing your intuition for others, you notice an error or gap in your reasoning that causes you to revisit your intuition – or make you question whether the statement is actually true!

A **mathematical proof** is how we communicate ideas about the truth or falsehood of a statement to others. There are many different philosophical ideas about what constitutes a proof, but what they all have in common is that a proof is a mode of *communication*, from the person creating the proof to the person digesting it. In this course, we will focus on reading and creating our own written mathematical proofs, which is the standard proof medium in computer science.

As with all forms of communication, the style and content of a proof varies depending on the audience. In this course, the audience for all of our proofs will be an average CSC165 student (and not your TA or instructor).

As we will discuss, your audience determines how formal your proof should be (here, quite formal), and what background knowledge you can assume is understood without explanation (here, not much).

Some basic examples

We're going to start out our exploration of proofs with a few simple statements. You may find them a bit on the easy side, which is fine. We are using them not so much for their ability to generate mathematical insight, but rather to model both the *thinking* and the *writing* that would go into approaching a problem.

Each example in this chapter is divided into three or four parts:

(i) The statement that we want to prove or disprove. Sometimes, we'll specify whether to prove or disprove it, and other times deciding whether the statement is true or false as part of the exercise.

(ii) A translation of the statement into predicate logic. This step often provides insight into the *logical structure* of the statement that we are considering, which in turn informs the structure and techniques that we will use in our proofs.

(iii) A discussion to try to gain some intuition about why the statement is true. You'll tend to see that these are written very informally, as if we are talking to a friend on a whiteboard. The discussion usually will reveal the mathematical insight that forms the content of a proof. **This is often the hardest part of developing a proof, so please don't skip these sections!**

(iv) A formal proof. This is meant to be a standalone piece of writing, the "final product" of our earlier work. Depending on the depth of the discussion, the formal proof might end up being almost mechanical – a matter of formalizing our intuition.

Example 2.1. Prove that $15 \cdot 3^2 - 7 = 7 + (19 + 3)^2/4$.

Translation. Note that this statement has no logical operators, variables, or quantifiers. So the "translation" into predicate logic is simply itself:

$$15 * 3^2 - 7 = 7 + (19 + 3)^2/4.$$

Discussion. I can check whether this is true or not by putting both sides into my calculator.

Proof. This statement is true because both sides equal 128. □

Note: we are not going to evaluate you on your computational abilities. We expect for a typical CSC165 student that you can check arithmetic expressions yourself, without having to see each subexpression evaluated one step at a time. You can have the same expectation when writing your proofs.

Example 2.2. There exists a power of two bigger than 1000.

Translation. In order to translate this statement into predicate logic, I need to unpack two definitions in this statement. I know that "there exists" translates into an existential quantifier, and all "powers of 2" have the form 2^n, where n is a natural number. So this statement becomes:

$$\exists n \in \mathbb{N}, 2^n > 1000.$$

Discussion. This must be true since I know that the powers of 2 grow to infinity (whatever that means). I just need to do some calculations until I find a large enough value for n.

Proof. Let $n = 10$. Then 2^n is a power of two, and $2^n = 1024$, which is bigger than 1000. □

Note again that we didn't add a sentence in our proof to "verify" the calculation that $2^{10} = 1024$, as this is something easily checkable with a calculator.

We can draw from this example a more general technique for structuring our existence proofs. A statement of the form $\exists x \in S, P(x)$ is true when some element of S satisfies P. The easiest way to convince someone that this is true is to actually find the concrete element that satisfies P, and then show that it does. This is so natural a strategy that it should not be surprising that there is a "standard proof format" when dealing with such statements.

Of course, this is *not* the only proof technique used for existence proofs. As you'll see in more advanced studies, there are other, more sophisticated ways of proving that an existentially-quantified statement is true.

A typical proof of an existential.

Given statement to prove: $\exists x \in S, P(x)$.

Proof. Let $x =$ _____.

[Proof that $P($_____$)$ is true.] □

Note that the two blanks _____ represent the same element, which *you* get to choose as a prover. Thus existence proofs usually come down to *finding* a correct element of the domain which satisfy the required properties.

Example 2.3. Every real number n bigger than 20 satisfies the inequality $1.5n - 4 \geq 3$.

Translation. Here the statement starts with an "every," and even gives a name to this quantified variable n. These are big hints about the structure of the statement: it is universally quantified.

What about the domain of n? The statement mentions real numbers, but there is the issue of the qualifying "bigger than 20" as well. While we could define a set S to be the set of real numbers bigger than 20, instead we will see how to express this statement *without* defining a new set.

The condition "bigger than 20" can be expressed as a hypothesis in an implication: the conclusion, $1.5n - 4 \geq 3$ only needs to be true when n is bigger than 20.

This gives us the full translation

$$\forall n \in \mathbb{R}, n > 20 \Rightarrow 1.5n - 4 \geq 3.$$

Discussion. I might first try to gain some intuition by substituting numbers for n. 25 is bigger than 20, and $1.5(25) - 4 = 33.5 > 3$. But that is limited in scope to just one real number, and here I'm talking about an infinite number of real numbers. I need to use an argument that will work on *any* real number bigger than 20.

This should be some straightforward algebraic manipulation. We start with the assumption that $n > 20$, and multiply by 1.5 then subtract 4 – both of these operations will preserve the inequality.

Proof. Let $n \in \mathbb{R}$ be an *arbitrary* real number. *Assume* that $n > 20$.

We can perform the following manipulations to this inequality:

$$\begin{aligned} n &> 20 \\ 1.5n &> 30 \\ 1.5n - 4 &> 26 \\ 1.5n - 4 &\geq 3 && \text{(since } 26 > 3\text{)} \end{aligned}$$

□

The above proof has a couple of interesting details. The first is that this was a proof of a universally-quantified statement. Unlike the previous example, where we proved a fact about just one number, here we proved a fact about an *infinite* set of numbers.

To do this, our proof introduced a variable n that didn't take on a fixed value, but rather could represent any number in that set. Unlike the previous existence proof, when we introduced this variable n, we did not specify a concrete value, but rather said that n was "an arbitrary

real number," and then proceeded with the proof. As we get more comfortable, we will drop the English phrase part and just write "let $n \in S$" to introduce n as an arbitrary element of S.

A typical proof of a universal.

Given statement to prove: $\forall x \in S, P(x)$.

Proof. Let $x \in S$.

[Proof that $P(x)$ is true]. □

However, this structure does not tell the full story. We also put a further restriction on n: "Assume that $n > 20$." Whenever we want to prove that an implication $P \Rightarrow Q$ is true, we do so by *assuming* that P is true, and then proving that Q must be true.

A typical proof of an implication (direct)

Given statement to prove: $P \Rightarrow Q$.

Proof. Assume P.

[Proof that Q is true.] □

Statements of the form $\forall n \in S, P(n) \Rightarrow Q(n)$ are probably the most common type of statements you'll prove. We combine the above two structures by both **introducing variables and stating all assumptions at the beginning of our proof**, followed by the meaty mathematics that forms the **content of our proof**.

A nice way of understanding what it means for n to be an *arbitrary* real number under the stated assumption is that we should be able to *substitute* any real number that satisfies the assumption ($n > 20$) into the body of the proof, and have the body still make sense:

We can perform the following manipulations to this inequality: substituting $n = 25$

$$\begin{aligned} 25 &> 20 \\ 1.5(25) &> 30 \\ 1.5(25) - 4 &> 26 \\ 1.5(25) - 4 &\geq 3 \qquad \text{(since } 26 > 3\text{)} \end{aligned}$$

However, the body does not necessarily make sense if we violate our assumption about n! Examine the last line carefully – what is it really saying?

We can perform the following manipulations to this inequality:

substituting $n = 4$

$$\begin{aligned} 4 &> 20 \\ 1.5(4) &> 30 \\ 1.5(4) - 4 &> 26 \\ 1.5(4) - 4 &\geq 3 \qquad \text{(since } 26 > 3\text{)} \end{aligned}$$

A note about inequalities, bounds, and approximation

You may have felt a little uneasy by the final step of our computation in the above proof, going from $1.5n - 4 > 26$ to $1.5n - 4 \geq 3$. In most calculations you would have done in high school (or perhaps even other university math classes), we never would have performed such a step. If we wanted to "solve" the inequality $1.5n - 4 \geq 3$, the "answer" we present would probably be $n \geq \frac{14}{3}$, not $n \geq 20$. What is different here?

We deliberately chose this example to bring up this point. There is a difference between *solving* an inequality to determine the exact range of values for a variable, and manipulating inequalities to produce more inequalities. Inequalities are fundamentally about *bounding* values, and are by definition inexact. In this course (and largely in computer science), we treat inequalities with a grain of salt, keeping in mind that they are just bounds.

When a bound is "as good as possible," we pay special attention to it: these bounds are not to be taken for granted, and must always be earned.

Of course, we'll see what we mean by "as good as possible" later on.

A new domain: number theory

One of the biggest questions that arises from "proof as communication" is determining how much detail to go into. For this course, we are assuming only basic knowledge of arithmetic, algebraic manipulations of equalities and inequalities, and standard elementary functions like powers, logarithms, and trigonometric functions. However, there is even variation in the typical CSC165 student with experience in this area, so as much as possible in this course, we will introduce *new mathematical domains* to serve as the objects of study in our proofs.

So you may use, without justifications, various laws like $a^b \cdot a^c = a^{b+c}$ and $\sin^2\theta + \cos^2\theta = 1$.

This approach has three very nice benefits: first, by building domains from the ground up, we can specify absolutely the common definitions and properties that everyone may assume and use freely in proofs; second, these domains are the theoretical foundation of many areas of computer science, and learning about them here will serve you well in many

future courses; and third, learning about new domains will help develop the skill of *reading about a new mathematical context and understanding it*. The definitions and axioms of a new domain communicate the foundation upon which we build new proofs – in order to prove things, we need to understand the objects that we're talking about first.

In other words, you won't just learn about new domains; you'll learn *how* to learn about new domains!

Our first foray into domain exploration will be into number theory, which you can think of as taking a type of entity with which we are quite familiar, and formalizing definitions and pushing the boundaries of what we actually know about these *numbers* that we use every day. We'll start off by repeating and expanding on one definition from the previous chapter.

Definition 2.1 (divisibility, divisor, multiple)**.** Let $n, d \in \mathbb{Z}$. We say that d **divides** n, or n **is divisible by** d, if and only if there exists a $k \in \mathbb{Z}$ such that $n = dk$.

In this case, we use the notation $d \mid n$ to represent "d divides n," and call d a **divisor** of n, and n a **multiple** of d.

Divisibility is a nice definition to work with because it contains an existential quantifier embedded in the definition. From this, we'll see some proofs with more complex structure, based on the greater complexity of the statement.

Later in this chapter, we'll look at some examples involving prime numbers, which is more complex.

Example 2.4. Prove that $23 \mid 115$.

Translation. We want to *expand* the definition of disibility to rewrite this statement in terms of simpler operators:

$$\exists k \in \mathbb{Z}, 115 = 23k.$$

Discussion. We just need to divide 115 by 23, right?

Proof. Let $k = 5$.

Then $115 = 23 \cdot 5$. □

Notice that this is still an instance of the "typical structure" for existential proofs.

Example 2.5. Prove that there exists an integer that divides 104.

Translation. There is the key phrase "there exists" right in the problem statement, so we could write $\exists a \in \mathbb{Z}, a \mid 104$. We can once again expand the definition of divisibility to write

$$\exists a, k \in \mathbb{Z}, 104 = ak.$$

Notice how we combine the existential quantifiers for a and k into a single abbreviated form.

Discussion. So we basically need to pick a pair of divisors of 104. Since this is an existential proof and we get to pick both a and k, any pair of divisors will work.

Proof. Let $a = -2$ and let $k = -52$.

Then $104 = ak$. □

This example is the first one that had multiple quantifiers. In our proof, we had to give explicit values for both a and k to show that the statement held. Just as how a *sentence* in predicate logic must have all of its variables quantified, a *mathematical proof* must introduce *all* variables contained in the sentence.

We typically introduce variables with the word "let" in a proof. But whether we let a variable be a concrete value or an arbitrary element of the domain depends on how the variable is quantified in the formula.

Alternating quantifiers revisited

In the previous chapter, you saw how changing the order of an existential and universal quantifier changed the meaning of a statement. In this section, we will consider how the order of quantifiers affects how we introduce variables in a proof.

Example 2.6. Prove that all integers are divisible by 1.

Translation. The statement contains a universal quantification: $\forall n \in \mathbb{Z}, 1 \mid n$. We can unpack the definition of divisibility to

$$\forall n \in \mathbb{Z}, \exists k \in \mathbb{Z}, n = 1 \cdot k.$$

Discussion. The final equation in the fully-expanded form of the statement is straightforward, and is valid when k equals n. But how should I introduce these variables? *In the same order they are quantified in the statement.*

Proof. Let $n \in \mathbb{Z}$. Let $k = n$.

Then $n = 1 \cdot k$. □

In this proof, we used an extremely important tool at our disposal when it comes to proofs with multiple quantifiers: **any existentially-quantified variable can take on a value in a proof that depends on the values of the variables which have been defined before it.**

First, we defined n to be an arbitrary integer. Immediately after this, we wanted to show that for this n, $\exists k \in \mathbb{N}, n = 1 \cdot k$. And to prove this, we needed a value for k – a "let" statement. Because we define k after having defined n, we can use n in the definition of k and say "Let $k = n$." It may be helpful to think about the analogous process in programming. We first define a variable n, and then define a new variable k that is assigned the value of n.

Even though this may seem obvious, one important thing to note is that the *order of variables in the statement determines the order in which the*

variables must be introduced in the proof, and hence which variables can depend on which other variables.

For example, consider the following erroneous "proof."

Example 2.7 (Wrong!)**.** Prove that $\exists k \in \mathbb{Z}, \forall n \in \mathbb{Z}, n = 1 \cdot k$.

Proof. Let $k = n$. Let n be an arbitrary integer.

Then $n = 1 \cdot k$. □

This proof may look very similar to the previous one, but it contains one crucial difference. The very first sentence, "Let $k = n$," is invalid: at that point, n has not yet been defined! This is the result of having switched around the order of the quantifiers, which forces k to be defined independently of whatever n is chosen.

Note: don't assume that just because *one* proof is invalid, that *all* proofs of this statement are invalid! So we can't conclude that this statement is false just because we found one proof that didn't work. We'll next look at how to prove that this statement is indeed false.

A meta way of looking at this: a statement is true if *there exists* a correct proof of the statement.

False statements, false proofs

Suppose we have a friend who is trying to convince us that a certain statement X is false. If they tell you that statement X is false because they tried really hard to come up with a proof of it and failed, you might believe them, or you might wonder if maybe they just missed a crucial idea leading to a correct proof. An absence of proof is not enough to convince us that the statement is false.

Maybe they skipped all their CSC165 classes, for example.

Instead, to be fully convinced that a statement is false, we must see a **disproof**, which is simply a proof that the *negation* of the statement is true. For this section, we'll be using the simplification rules from the first chapter to make negations of statements easier to work with.

In other words, if we can prove that $\neg P$ is true, then P must be false.

Here are two examples: the first one is extremely simple, and is used to introduce the basic idea. The second is more subtle, and really requires good understanding of how we manipulate a statement to get a simple form for its negation.

Example 2.8. Disprove the following statement: every natural number divides 360.

Translation. This statement can be written as

$$\forall n \in \mathbb{N}, n \mid 360.$$

However, we want to prove that it is false, so we really need to study its negation.

$$\neg(\forall n \in \mathbb{N}, n \mid 360)$$
$$\exists n \in \mathbb{N}, n \nmid 360$$

Discussion. The original statement is obviously not true: the number 7 doesn't divide 360, for instance. Is that a proof? We wrote the negation of the statement in symbolic form above, and if we translate it back into English, we get "there exists a natural number which does not divide 360." So, yes. That's enough for a proof.

Proof. Let $n = 7$.

Then $n \nmid 360$, and so not every natural number divides 360. □

When we want disprove a universally quantified statement ("every element of S satisfies property P"), the negation of that statement becomes an existentially quantified one ("there exists an element of S that doesn't satisfy property P"). Since proofs of existential quantification involve just finding one value, the disproof of the original statement involves finding such a value which causes the property to be false (or alternatively, causes the negation of the property to be true). We call this value a **counterexample** for the original statement, a term you might already be familiar with.

We would say that 7 is a counterexample of the statement "every natural number divides 360."

A typical disproof of a universal (counterexample).

Given statement to *disprove*: $\forall x \in S, P(x)$.

Proof. We prove the negation, $\exists x \in S, \neg P(x)$.

Let $x =$_____.

[Proof that $P($_____$)$ is *false*.] □

Example 2.9. Disprove the following claim: for all natural numbers a and b, there exists a natural number c which is less than $a + b$, and greater than both a and b, such that c is divisible by a or by b.

Translation. The original statement can be translated as follows. We've underlined the four different propositions which are joined with **AND**s to make them stand out.

$$\forall a, b \in \mathbb{N}, \exists c \in \mathbb{N}, \underline{c < a + b} \wedge \underline{c > a} \wedge \underline{c > b} \wedge \underline{(a \mid c \vee b \mid c)}.$$

We'll derive the negation step by step, though once you get comfortable with the negation rules, you'll be able to handle even complex formulas like this one quite quickly.

$$\neg\Big(\forall a, b \in \mathbb{N}, \exists c \in \mathbb{N}, \underline{c < a + b} \wedge \underline{c > a} \wedge \underline{c > b} \wedge \underline{(a \mid c \vee b \mid c)}\Big)$$

$$\exists a, b \in \mathbb{N}, \neg\Big(\exists c \in \mathbb{N}, \underline{c < a + b} \wedge \underline{c > a} \wedge \underline{c > b} \wedge \underline{(a \mid c \vee b \mid c)}\Big)$$

$$\exists a, b \in \mathbb{N}, \forall c \in \mathbb{N}, \neg\Big(\underline{c < a + b} \wedge \underline{c > a} \wedge \underline{c > b} \wedge \underline{(a \mid c \vee b \mid c)}\Big)$$

$$\exists a, b \in \mathbb{N}, \forall c \in \mathbb{N}, \underline{c \geq a + b} \vee \underline{c \leq a} \vee \underline{c \leq b} \vee \underline{\Big(\neg(a \mid c \vee b \mid c)\Big)}$$

$$\exists a, b \in \mathbb{N}, \forall c \in \mathbb{N}, \underline{c \geq a + b} \vee \underline{c \leq a} \vee \underline{c \leq b} \vee \underline{(a \nmid c \wedge b \nmid c)}$$

Discussion. That symbolic negation involved quite a bit of work. Let's make sure we can translate the final result back into English: there exist natural numbers a and b such that for all natural numbers c, $c \geq a + b$ or $c \leq a$ or $c \leq b$ or neither a nor b divide c. Hopefully this example illustrates the power of predicate logic: by first translating the original statement into symbolic logic, we were able to obtain a negation by applying some standard manipulation rules and then translating the resulting statement back into English. For a statement as complex as this one, it is usually easier to do this than to try to "figure out" what the English negation of the original is, at least when you're first starting out.

Okay, so how do we prove the negation? The existential quantifier tells us we get to pick a and b. Let's think simple: what if a and b are both 2? Then $a + b = 4$. If $c \geq 4$, the first clause in the OR is satisfied, and if $c \leq 2$, the second and third clauses are satisfied. So we only need to worry about when c is 3, because in this case the only clause that could possibly be satisfied is the last one, $a \nmid c \wedge b \nmid c$. Luckily, a and b are both 2, and 2 doesn't divide 3, so it seems like we're good in this case as well.

It was particularly helpful that we chose such small values for a and b, so that there weren't a lot of numbers in between them and their sum to care about. As you do your own proofs of existentially-quantified statements, remember that you have the power to pick values for these variables!

Proof. Let $a = 2$ and $b = 2$, and let $c \in \mathbb{N}$. We now need to prove that

$$c \geq a + b \vee c \leq a \vee c \leq b \vee (a \nmid c \wedge b \nmid c).$$

Substituting in the values for a and b, this gets simplified to:

$$c \geq 4 \vee c \leq 2 \vee 2 \nmid c \qquad (*)$$

To prove an OR, we only need one of the three parts to be true, and different ones can be true for different values of c.

However, precisely which part is true depends on the value of c. For example, we can't say that for an *arbitrary* value of c, that $c \geq 4$. So we'll split up the remainder of the proof into three cases for the values for c: numbers ≥ 4, numbers ≤ 2, and the single value 3.

Note that *every* natural number falls into one of these cases.

Case 1. In this case, we will *assume* that $c \geq 4$, and prove the statement $(*)$ is true.

Then the first part is true.

Case 2. In this case, we will *assume* that $c \leq 2$, and prove the statement $(*)$ is true.

In this case, the second part is true.

Case 3. In this case, we will *assume* that $c = 3$, and prove the statement $(*)$ is true.

This case is the trickiest, because unlike the other, our assumption that $c = 3$ is not verbatim one of the parts of $(*)$. However, we note that $2 \nmid 3$, and so the third part is satisfied.

Since in all possible cases statement $(*)$ is true, we conclude that this statement is always true. □

Proof by cases

The previous proof illustrated a new proof technique known as **proof by cases**. Remember that for a universal proof, we typically let a variable be an arbitrary element of the domain, and then make an argument in the proof body to prove our goal statement. However, even when the goal statement is true for all elements of the domain, it isn't always easy to construct an argument that works for all elements of the domain! Sometimes, different arguments are required for different elements. In this case, we divide the domain into different parts, and then write a separate argument for each part separately.

Intuitively, the statement is true for different elements for different reasons.

A bit more formally, we pick a set of unary predicates P_1, P_2, ..., P_k (for some positive integer k), such that for every element x in the domain, x satisfies at least one of the predicates (we say that these predicates are *exhaustive*).You should think of these predicates as describing how we divide up the domain; in the previous example, the predicates were:

$$P_1(c) : c \leq 2, \qquad P_2(c) : c \geq 4, \qquad P_3(c) : c = 3.$$

Then, we divide the proof body into cases, where in each case we *assume* that one of the predicates is True, and us that to construct a proof that specifically works under that assumption.

A typical proof by cases.

Given statement to prove: $\forall x \in S, P(x)$.

Pick a set of exhaustive predicates $P_1, \ldots, P_k$.

Proof. Let $x \in S$.

Case 1. *Assume* $P_1(x)$ is true.

[Proof that $P(x)$ is true, assuming $P_1(x)$.]

Case 2. *Assume* $P_2(x)$ is true.

[Proof that $P(x)$ is true, assuming $P_2(x)$.]

$\vdots$

Case k. *Assume* $P_k(x)$ is true.

[Proof that $P(x)$ is true, assuming $P_k(x)$.] □

Proof by cases is a very versatile proof technique, since it allows the combining of simpler proofs together to form a whole proof. Often it is easier to prove a property about some (or even most) elements of the domain than it is to prove that same property about all the elements. But do keep in mind that if you can find a *simple* proof which works for all elements of the domain, that's generally preferable than combining multiple proofs together in a proof by cases.

Generalizing statements

In this section, we will investigate another important skill for reading and writing proofs: the ability to *generalize* existing knowledge into more generic, and powerful, forms. As usual, we start with an example.

Example 2.10. Prove that for all integers x, if x divides $(x + 5)$, then x also divides 5.

Translation. There is both a universal quantification and implication in this question: $\forall x \in \mathbb{Z}, x \mid (x + 5) \Rightarrow x \mid 5$. When we unpack the defini-

We weren't kidding when we said this is the most common form of statement.

tion of divisibility, we need to be careful about how the quantifiers are grouped:

$$\forall x \in \mathbb{Z}, \Big(\big(\exists k_1 \in \mathbb{Z}, x + 5 = k_1 x\big) \Rightarrow \big(\exists k_2 \in \mathbb{Z}, 5 = k_2 x\big) \Big).$$

Discussion. So I need to prove that if x divides $x + 5$, then it also divides 5. So I can *assume* that x divides $x + 5$, and I need to *prove* that x divides 5. Since x is divisible by x, I should be able to subtract it from $x + 5$ and keep the result a multiple of x. Can I prove that using the definition of divisibility? I basically need to "turn" the equation $x + 5 = k_1 x$ into the equation $5 = k_2 x$.

Proof. Let x be an arbitrary integer. *Assume* that $x \mid (x + 5)$, i.e., that there exists $k_1 \in \mathbb{Z}$ such that $x + 5 = k_1 x$. Let k_1 be this value.

We want to prove that there exists $k_2 \in \mathbb{Z}$ such that $5 = k_2 x$. Let $k_2 = k_1 - 1$.

Then we can calculate:

$$\begin{aligned} k_2 x &= (k_1 - 1)x \\ &= k_1 x - x \\ &= (x + 5) - x && \text{(We assumed } x + 5 = k_1 x\text{)} \\ &= 5 \end{aligned}$$

□

Whew, that was a a bit longer than the proofs we've already done. There were a lot of new elements that we introduced here, so let's break them down:

- After introducing x, we wanted to prove the *implication* $x \mid (x + 5) \Rightarrow x \mid 5$. To prove an implication, we needed to assume that the hypothesis was true, and then prove that the conclusion is also true. In our proof, we wrote "**Assume** $x \mid (x + 5)$." The goal for the rest of the proof after that was to prove that $x \mid 5$.

  Note that this proof did **not** prove that $\forall x \in \mathbb{Z}, x \mid x + 5$: this is actually very false! Instead, we proved that *if* x divides $x + 5$, *then* it must also divide 5.
- When we assumed that $x \mid (x + 5)$, what this really did was introduce a new variable $k_1 \in \mathbb{Z}$ from the definition of divisibility. This might seem a little odd, but take a moment to think about what this means in English. We assumed that x divides $x + 5$, which (by definition) is

the same as assuming that there exists an integer k_1 such that $x + 5 = k_1 x$. Given that such a number exists, we can give it a name and refer to it in the rest of our proof!

In other words, we introduced a variable into the proof through an assumption we made. We can see this coming immediately from the fully-expanded logical statement.

One of the most important meta-techniques in mathematical proof is that of **generalization**: taking a true statement (and a proof of the statement), and then replacing a concrete value in the statement with a universally quantified variable. For example, consider the statement from the previous example, $\forall x \in \mathbb{Z}, x \mid (x+5) \Rightarrow x \mid 5$. It doesn't seem like the "5" serves any special purpose; it is highly likely that it could be replaced by another number like 165, and the statement would still hold.

Concretely, consider the statement $\forall x \in \mathbb{Z}, x \mid (x+165) \Rightarrow x \mid 165$, which is at least as plausible as the original statement with 5's.

But rather than replace the 5 with another concrete number and then re-proving the statement, we will instead replace it with a universally quantified variable, and prove the corresponding statement. This way, we will know that in fact we could replace the 5 with *any* natural number, and the statement would still hold.

Example 2.11. Prove that for all $d \in \mathbb{Z}$, for all $x \in \mathbb{Z}$, if x divides $(x+d)$, then x also divides d.

Translation. This has basically the same translation as last time, except now we have an extra variable:

$$\forall d, x \in \mathbb{Z}, \Big(\big(\exists k_1 \in \mathbb{Z}, x + d = k_1 x\big) \Rightarrow \big(\exists k_2 \in \mathbb{Z}, d = k_2 x\big) \Big).$$

Discussion. I should be able to use the same set of calculations as last time.

Proof. Let d and x be arbitrary natural numbers. *Assume* that $x \mid (x+d)$, i.e., there exists $k_1 \in \mathbb{Z}$ such that $x + d = k_1 x$.Let k_1 be this value.

We want to prove that there exists $k_2 \in \mathbb{Z}$ such that $d = k_2 x$. Let $k_2 = k_1 - 1$.

Then we can calculate:

$$\begin{aligned} k_2 x &= (k_1 - 1)x \\ &= k_1 x - x \\ &= (x + d) - x && \text{(We assumed } x + d = k_1 x\text{)} \\ &= d \end{aligned}$$

□

This proof is basically the same as the previous one: we have simply swapped out all of the 5's with d's. We say that the proof *did not depend on the value 5*, meaning there was no place that we used some special property of 5, where we could have used a generic integer instead. We can also say that the original statement and proof *generalize* to this second version.

Why does generalization matter? By generalizing the previous statement from being about the number 5 to an arbitrary integer, we have essentially gone from one statement being true to an infinite number of statements being true. The more general the statement, the more useful it becomes. We care about exponent laws like $a^b \cdot a^c = a^{b+c}$ precisely because they apply to every possible number; regardless of what our concrete calculation is, we know we can use this law in our calculations.

Exercise Break!

2.1 Prove that for any three integers a, b, and c, if a divides both b and c, then a also divides $b + c$.

Hint: since the hypothesis is an **AND** of two statements, you get to assume two statements.

2.2 Generalize the previous proof to prove the following statement:

$$\forall a, b, c, p, q \in \mathbb{Z}, \Big(a \mid b \wedge a \mid c \Rightarrow a \mid (bp + cq)\Big).$$

This statement says that if you have two multiples of a, and then multiply them by any other two numbers and add the results, the final number must always be a multiple of a.

Or, the property of "being a multiple of a" is preserved by multiplication by any number, and adding any multiple of a.

Proof by contrapositive

To wrap up this section, let us look at one final example that is very similar to the previous one.

Example 2.12. Prove that for all integers x, if x does not divide $x + 5$, then x does not divide 5.

Translation. This is actually a little easier to translate than the examples we have just done. We'll keep the $\mid$ predicate in the statement for now.

$$\forall x \in \mathbb{Z}, x \nmid x + 5 \Rightarrow x \nmid 5.$$

Discussion. As a standard approach for an implication, we would first assume that x does not divide $x + 5$, and then prove that x does not divide 5. But assuming that x *doesn't* divide something seems less informative than knowing that it does divide something.

Luckily, we have new proof technique to work with: an **proof by contrapositive** (also known as a form of indirect proof). Rather than try to prove the implication directly, we prove its contrapositive, which is logically equivalent to it. Let's rewrite the statement using the contrapositive:

Remember, the contrapositive of $P \Rightarrow Q$ is $\neg Q \Rightarrow \neg P$.

$$\forall x \in \mathbb{Z}, x \mid 5 \Rightarrow x \mid x + 5.$$

Now if we can assume $x \mid 5$, that gives us a lot to work with!

Proof. Let $x \in \mathbb{Z}$. We will prove the contrapositive statement: $x \mid 5 \Rightarrow x \mid x + 5$. So assume that $x \mid 5$.

[We leave it as an exercise to prove that $x \mid x + 5$ under this assumption.] □

When proving an implication, it is often the case that the assuming the hypothesis does not get you very far. Flipping the implication around to its contrapositive and assuming the negation of the conclusion might yield better results!

A typical proof of an implication (contrapositive/indirect)

Given statement to prove: $P \Rightarrow Q$.

Proof. Assume $\neg Q$.

[Proof that $\neg P$ is true.] □

Greatest common divisor

Let us now introduce one more definition that you're probably familiar with, though again we will take some time to treat it more formally than what you may have seen before.

Definition 2.2 (greatest common divisor)**.** Let m, n be natural numbers which are not both 0. The **greatest common divisor (gcd)** of m and n, denoted $\gcd(m, n)$, is the maximum natural number d such that d divides both m and n.

According to this definition, what is $\gcd(0, n)$ when $n > 0$?

We also define $\gcd(0,0) = 0$ just to make the domain of the gcd operator all possible pairs of natural numbers.

To make it easier to translate this statement into symbolic form, we can restate the "maximum" part by saying that if e is any number which divides m and n, then $e \leq d$. Let $m, n, k \in \mathbb{N}$, not all of which are 0, and suppose $k = \gcd(m,n)$. Then k satisfies the following formula:

$$k \mid m \wedge k \mid n \wedge \big(\forall e \in \mathbb{N}, e \mid m \wedge e \mid n \Rightarrow e \leq k\big).$$

You might wonder whether this definition makes sense in all cases: is it possible for two numbers to have no divisors in common? But remember that one of the statements we proved in this chapter is that 1 divides every natural number. So at the very least, 1 is a common divisor between any two natural numbers.

Here is an example which makes use of both this definition, and the definition of **prime** from the previous chapter.

Example 2.13. Let p and q be positive integers, and assume that p and q are distinct primes. Then $\gcd(p,q) = 1$.

Note that our use of "let" here is to conveniently introduce name of variables to refer to later in the theorem. It is possible to rewrite this whole statement using the English phrases "for all," but this is rather cumbersome.

Translation. Here is an initial translation which focuses on the structure of the above statement, but doesn't unpack any definitions:

$$\forall p, q \in \mathbb{Z}^+, (Prime(p) \wedge Prime(q) \wedge p \neq q) \Rightarrow \gcd(p,q) = 1.$$

We could unpack the definitions of *Prime* and gcd, but doing so does not add any insight at this point. While we will almost certainly end up using these definitions in the discussion and proof sections, expanding it here actually obscures the meaning of the statement.

In general, use translation as a way of precisely specifying the *structure* of a statement; as we have seen repeatedly, the high-level structure of a statement is mimicked in the structure of its proof. And while you don't need to expand every definition in a statement, you should *always* keep in mind that definitions referred to in the statement will require unpacking in the proof itself.

Discussion. We know that primes don't have many divisors, and that 1 is a common divisor for any pair of numbers. So to show that $\gcd(p,q) = 1$, we just need to make sure that neither p nor q divides the other (otherwise that would be a larger common divisor than 1).

Proof. Let $p, q \in \mathbb{Z}^+$. Assume that p and q are both prime, and that $p \neq q$. We want to prove that $\gcd(p,q) = 1$.

By the definition of primality, the only positive divisors of p are 1 and p, and the only divisors of q are 1 and q. So then since $p \neq q$ and $p \neq 1$, we know that $p \nmid q$.

Then 1 is the only positive common divisor of p and q, so $\gcd(p, q) = 1$. □

Next, we will look at one of the strongest properties of the greatest common divisor: it is the smallest natural number that can be written as a sum of (positive and negative) multiples of the two numbers.

Theorem 2.1. *Let a and b be arbitrary natural numbers which are not both zero. Then $\gcd(a, b)$ is the smallest natural number such that there exist $p, q \in \mathbb{Z}$ with $\gcd(a, b) = ap + bq$.*

We will not prove this theorem here; instead, our main goal for stating it is to introduce a new proof technique: using an external statement as a step in a proof. This might sound kind of funny – after all, many of our proofs so far have relied on some algebraic manipulations which are valid but are really knowledge we learned prior to this course. The subtle difference is that those algebraic laws we take for granted as "obvious" because we learned them so long ago. But in fact our proofs can consist of steps which are statements that we know are true because of an external source, even one that *we don't know how to prove ourselves*.

This is a fundamental parallel between writing proofs and writing computer programs. In programming, we start with some basic building blocks of a language – data types, control flow constructs, etc. – but we often rely on libraries as well to simplify our tasks. We can use these libraries by reading their documentation and understanding how to use them, but don't need to understand how they are implemented. In the same way, we can use an external theorem in our proof by understanding what it means, but without knowing how to prove it.

Example 2.14. Let $a, b \in \mathbb{N}$. Every integer that divides both a and b also divides $\gcd(a, b)$.

Translation.

$$\forall a, b \in \mathbb{N}, \forall d \in \mathbb{Z}, (d \mid a \wedge d \mid b) \Rightarrow d \mid \gcd(a, b).$$

Discussion. This one is a bit tougher. All we know from the definition of gcd is that $d \leq \gcd(a, b)$, but that doesn't imply $d \mid \gcd(a, b)$ by any means. But given the context that we just discussed in the preceding paragraphs, I'd guess that we should also use Theorem 2.1 to write $\gcd(a, b)$ as $ap + bq$. Oh, and one of the previous exercises showed that any number that divides a and b will divide $ap + bq$ as well!

Proof. Let $a, b \in \mathbb{N}$ and $d \in \mathbb{Z}$. Assume that $d \mid a$ and $d \mid b$. We want to prove that $d \mid \gcd(a, b)$.

By Theorem 2.1, there exist integers $p, q \in \mathbb{Z}$ such that $\gcd(a, b) = ap + bq$. Let p and q be such integers.

Then by Exercise 2.2, since $d \mid a$ and $d \mid b$ (by assumption), we know that $d \mid ap + bq$. Since $\gcd(a, b) = ap + bq$, we conclude that $d \mid \gcd(a, b)$. □

Modular arithmetic

The final definition in this chapter introduces some notation that is extremely commonplace in number theory, and by extension in many areas of computer science. Often when we are dealing with relationships between numbers, divisibility is too coarse a relationship: as a predicate, it is constrained by the binary nature of its output. Instead, we often care about the *remainder* when we divide a number by another.

Definition 2.3 (modular congruence)**.** Let $a, b, n \in \mathbb{Z}$, with $n \neq 0$. We say that a **is congruent to** b **modulo** n if and only if $n \mid a - b$. In this case, we write $a \equiv b \pmod{n}$.

One warning: the notation $a \equiv b \pmod{n}$ is *not* exactly the same as the `mod` or `%` operator you are familiar with from programming; here, both a and b could be much larger than n, or even negative.

This definition captures the idea that a and b have the *same remainder* when divided by n. You should think of this congruence relation as being analogous to numeric equality, with a relaxation. When we write $a = b$, we mean that the numeric values of a and b are literally equal. When we write $a \equiv b \pmod{n}$, we we mean that if you look at the remainders of a and b when divided by n, those remainders are literally equal.

We will next look at how addition, subtraction, and multiplication all behave in an analogous fashion under modular arithmetic. The following proof is a little tedious because it is calculation-heavy; the main benefits here are practicing reading and using a new definition, and getting comfortable with this particular notation.

Example 2.15. Prove that for all $a, b, c, d, n \in \mathbb{Z}$, with $n \neq 0$, if $a \equiv c \pmod{n}$ and $b \equiv d \pmod{n}$, then:

(1) $a + b \equiv c + d \pmod{n}$

(2) $a - b \equiv c - d \pmod{n}$

(3) $ab \equiv cd \pmod{n}$

Translation. We will only show how to unpack the definitions in (1), as the other two are quite similar.

$$\forall a, b, c, d, n \in \mathbb{Z}, \big(n \neq 0 \wedge n \mid (a-c) \wedge n \mid (b-d)\big) \Rightarrow n \mid \big((a-b)-(c-d)\big).$$

Proof. Let $a, b, c, d, n \in \mathbb{N}$, and assume that $n \neq 0$, $n \mid (a-b)$, and $n \mid (b-d)$.

We will only prove (1), and leave (2) and (3) as exercises. This means we want to prove that $n \mid ((a-b)-(c-d))$.

By Exercise 2.2, since $n \mid (a-c)$ and $n \mid (b-d)$, it divides their difference:

$$\begin{aligned} &n \mid (a-c)-(b-d) \\ &n \mid (a-b)-(c-d) && \text{(rearranging terms)} \end{aligned}$$

□

You may be wondering why we left out division in the above theorem. Recall again the definition of divisibility: $a \mid b$ means that there exists $k \in \mathbb{N}$ such that $b = ka$. Not every pair of integers is related by divisibility, and this also transfers over to modular arithmetic as well.

Division over integers is different than division over the real numbers, in other words.

However, we have all the tools necessary to prove the following quite remarkable fact.

Example 2.16. Let $a, b, p \in \mathbb{Z}$. If p is a prime number and a is not divisible by p, then there exists $k \in \mathbb{Z}$ such that $ak \equiv b \pmod{p}$.

Translation. This statement is quite complex! Remember that we focus on translation to examine the structure of the statement, so that we know how to set up a proof. We aren't going to expand every single definition for the sake of expanding definitions.

$$\forall a, b, p \in \mathbb{Z}, \Big(\big(Prime(p) \wedge p \nmid a\big) \Rightarrow \big(\exists k \in \mathbb{Z}, ak \equiv b \pmod{p}\big)\Big).$$

Discussion. So this is saying that under the given assumptions, b is "divisible" by a modulo p. Somehow I'm supposed to use the fact that p is prime. The conclusion is "there exists a $k \in \mathbb{Z}$ such that..." so that I know that at some point I'll need to define a variable k in terms of a, b, and/or p, which satisfies the congruence.

Can I do $k = b/a$? That obviously would satisfy the congruence, but the example statement doesn't say that I can assume that a divides b... But if I could prove that $a \mid b$, then I would be able to write the proof.

So is it true? The statement has to hold for *every* pair of numbers a and b where a isn't divisible by p, so I think I'm out of luck – after all, this includes cases where $a > b$.

Here's another idea: can I prove a *less general* statement? I could set b to always be 1, and try to show that there always exists a k such that $ak \equiv 1 \pmod{p}$. If I can show that, then multiplying both sides by b should do the trick.

That's statement (3) from the previous example, by the way.

[**HINT**: use Theorem 2.1.] Woah, I got a hint! Hmmm, Theorem 2.1 talks about writing gcd as a sum of multiples. How does that help? Let me write down what I know and can assume:

- p is prime
- $p \nmid a$
- The gcd of two numbers can be written as the *sum of multiples* of the numbers.

And what I want to prove:

- $\exists k \in \mathbb{Z}, ak \equiv 1 \pmod{p}$. That's equivalent to:
- $\exists k \in \mathbb{Z}, p \mid (ak - 1)$, using the definition of mod. That's equivalent to:
- $\exists k, d \in \mathbb{Z}, ak - 1 = pd$. Hey, wait a second...
- $\exists k, d \in \mathbb{Z}, ak - pd = 1$. That's writing 1 as a *sum of multiples* of a and p!

Now I just need to connect these two lines of reasoning.

Proof. Let $a, b, p \in \mathbb{N}$. Assume that p is prime and p does not divide a. We want to prove that there exists $k \in \mathbb{Z}$ such that $ak \equiv b \pmod{p}$. To do this, we are going to first prove two *subclaims*.

Think of these as helper functions in programming. They are smaller proofs of smaller statements which we can use inside a larger proof.

Claim 1. $\gcd(a, p) = 1$.

Proof of Claim 1. By definition of **prime**, we know that the only two positive divisors of p are 1 and p. Since we have assumed that $p \nmid a$, this means that 1 is the only positive common divisor of p and a. So $\gcd(a, p) = 1$. □

Claim 2. There exists $k \in \mathbb{Z}$ such that $ak \equiv 1 \pmod{p}$.

Proof. By the previous claim, we now know that $\gcd(a, p) = 1$. By Theorem 2.1, there exist $r, s \in \mathbb{Z}$ such that $ar + ps = 1$.

Let $k = r$. Then we can re-arrange this statement:

$$\begin{aligned} ak + ps &= 1 \\ ak - 1 &= p(-s) \\ p \mid (ak - 1) & \qquad \text{(definition of divisibility)} \\ ak \equiv 1 \pmod{p} & \qquad \text{(definition of congruence)} \end{aligned}$$

□

Finally, we can use these two claims to prove that there exists a $k' \in \mathbb{Z}$ such that $ak' \equiv b \pmod{p}$.

Let $k' = kb$. Then we have:

$$\begin{aligned} ak &\equiv 1 \pmod{p} \\ akb &\equiv b \pmod{p} \qquad \text{(multiply both sides by } b\text{)} \\ ak' &\equiv b \pmod{p} \end{aligned}$$

□

This theorem brings together elements from all of our study of proofs so far. We have both types of quantifiers, as well as some significant assumptions (as part of an implication). We even used Theorem 2.1 for a key step in our proof. Finally, this proof introduced one more useful kind of structure: a subproof, or proof of a smaller claim that is used to prove the main result. Just as helper functions help organize a program, small claims and subproofs help organize a proof so that each part can be understood separately, before being combined into a whole. As your proofs grow longer and longer, make good use of this approach to keep your proofs readable and easy to understand. There is nothing worse than having to slog through pages and pages of a single proof without any sense of what claim is being proved, and how the claims fit together.

We can outline the previous proof in three steps:

1. Prove that $\gcd(a, p) = 1$.
2. Prove that $\exists k \in \mathbb{Z}, ak \equiv 1 \pmod{p}$.
3. Prove that $\exists k' \in \mathbb{Z}, ak' \equiv b \pmod{p}$.

Proof by contradiction

The final proof technique we will cover in this chapter is the **proof by contradiction**. Given a statement P to prove, rather than attempt to prove it directly we assume that its *negation* $\neg P$ is True, and then use this assumption to prove a statement Q *and* its negation $\neg Q$. We call Q and $\neg Q$ the *contradiction* that arises from the assumption that P is False.

Why does this work? Essentially, we argue the *if* P is False, then statement Q must be True, but its negation $\neg Q$ must also be True. But these two things can't be true at the same time, and so our original assumption must be wrong!

Proofs by contradiction are a more general form of the indrect proof-by-contrapositive we saw earlier in this chapter. They often take a bit more thought because it isn't necessarily clear what the contradiction (statement Q) should be. We finish off this chapter by presenting one particularly famous proof by contradiction dating back to the Greek mathematician Euclid.

Although Euclid's original proof was written in a very informal style, the idea was certainly there!

Theorem 2.2. *There are infinitely many primes.*

Proof. Assume that this statement is False, i.e., that there a finite number of primes. Let $k \in \mathbb{N}$ be the number of primes, and let $p_1, p_2, \ldots, p_k$ be the prime numbers.

Our statement Q will be "for all $n \in \mathbb{N}$, n is prime if and only if n is one of $\{p_1, \ldots, p_k\}$." Q is True because of our assumption that there are a finite number of primes, and the definitions of k and $p_1, \ldots, p_k$.

Now we will show that Q is False. Define the number

$$P = 1 + \prod_{i=1}^{k} p_i = 1 + p_1 \times p_2 \times \cdots \times p_k.$$

There must be some prime p that divides P. But $p \notin \{p_1, \ldots, p_k\}$, because otherwise p would divide $P - p_1 \times \cdots \times p_k = 1$, and no prime can divide 1. So then p is a prime that is not one of $\{p_1, \ldots, p_k\}$, and so Q is false. Contradiction! □

We're using Exericse 2.2 here!

3 Induction

In the previous chapters we have studied how to express statements precisely using mathematical expressions, and how to analyze and prove the truth or falsehood of these statements using a variety of proof techniques, and with number theory as our primary domain. In this chapter, we will introduce a new and very important proof technique called **induction**, and use it to prove that some statement $P(n)$ is true for all values of n.

You may wonder why we need this new technique when we were already proving universal statements in the last chapter just fine without induction. It turns out that many interesting statements in number theory and most other domains cannot be proven or disproven with just the techniques from the previous chapter. We will first motivate the principle of induction using an example from modular arithmetic. Then we will apply induction to other statements in number theory, and then to new domains, using induction to prove properties about sequences and to find expressions for various ways of counting combinatorial objects.

The principle of induction

Let us start with an example.

Example 3.1. For any $m, x, y, n \in \mathbb{N}$ such that $n \geq 1$, if $x \equiv y \pmod{m}$, then $x^n \equiv y^n \pmod{m}$.

This is part (3) of Example 2.15 from the previous chapter.

It is not hard to show that this is true without using induction for $n = 2$ as follows. By assumption, $x \equiv y \pmod{m}$, and therefore $x \cdot x \equiv y \cdot y \pmod{m}$, and thus $x^2 \equiv y^2 \pmod{m}$. In order to show that it is true for $n = 3$, we can argue that since we already know that $x^2 \equiv y^2 \pmod{m}$, and $x \equiv y \pmod{m}$, then $x \cdot x^2 \equiv y \cdot y^2 \pmod{m}$ and thus $x^3 \equiv y^3 \pmod{m}$. Then we can prove that it is true for $n = 4$ in exactly the same way, and so on. But in order to make the "and so on" mathematically rigorous, we need to use induction.

The first explicit formulation of the principle of induction was given by Pascal (as in Pascal's triangle) in 1665. However, its uses have been traced as far back as Plato (370 BC), and a variation of Euclid's proof of the existence of infinitely many primes (from around the same time period). We cannot stress enough how important the induction principle is – it is *the* powerhorse behind nearly all proofs.

The principle of induction applies to *universal* statements – that is, statements of the form $\forall n \in \mathbb{N},\ P(n)$. It cannot be used to prove statements of any other form! Note however that $P(n)$ can be quite complicated and can involve other possibly nested quantifiers.

In this course, we will study only the most basic form of induction, commonly called *simple* induction. There are two steps to using this induction principle:

In CSC236, you'll learn about different forms of induction.

- The **base case** is a proof that the statement holds for the first natural number $n = 0$; that is, a proof that $P(0)$ holds.
- The **inductive step** is a proof that for all $k \in \mathbb{N}$, if $P(k)$ is true, then $P(k+1)$ is also true. That is:

$$\forall k \in \mathbb{N},\ P(k) \Rightarrow P(k+1).$$

Once the base case and inductive step are proven, by the principle of induction, one can conclude $\forall n \in \mathbb{N},\ P(n)$.

Typical structure of a proof by induction.

Given statement to prove: $\forall n \in \mathbb{N},\ P(n)$.

Proof. We prove this by induction on n.

Base Case: $n = 0$.

[Proof that $P(0)$ is true.]

Inductive Step: Let $k \in \mathbb{N}$, and *assume* that $P(k)$ is true.

[Proof that $P(k+1)$ is true.] □

The assumption that $P(k)$ is true is called the *induction hypothesis*.

The point behind induction is that sometimes it isn't possible to give a direct proof for all n – sometimes it requires knowing that the statement is true for smaller values in order to show that it is true for larger ones. Induction formalizes this idea – if you show it is true for the smallest element (the base case) and if you can show that as long as it is true for n then it is also true for the element right after n, then we can conclude that it is true for every n.

Why does the principle of induction work? This is essentially the domino effect. Assume you have shown the base case and the inductive step. In other words, you know $P(0)$ is true, and you know that $P(k)$ implies $P(k+1)$ for every natural number k. Since you know $P(0)$ from the base case and $P(0) \Rightarrow P(1)$ by the inductive step, we have $P(1)$. Then since you now know $P(1)$ and $P(1) \Rightarrow P(2)$ from the inductive step, we have $P(2)$. Now since we know $P(2)$ and $P(2) \Rightarrow P(3)$, we have $P(3)$. And so on.

Examples from number theory

Let us see how to use induction to prove some statements from number theory.

Example 3.2. For every natural number n, $7 \mid 8^n - 1$.

Translation. We can write this as

$$\forall n \in \mathbb{N},\ 7 \mid 8^n - 1.$$

Define the predicate $P(n)$ as "$7 \mid 8^n - 1$," where n is a natural number. This makes it clear how we will use induction: the statement becomes $\forall n \in \mathbb{N},\ P(n)$.

You'll see us start to merge the "translation" and/or "discussion" sections into the proof in this and future chapters, as you become more and more experienced with reading and writing proofs.

Proof. Let $P(n)$ be the statement that 7 divides $8^n - 1$; in other words, there exists an integer y such that $7 \cdot y = 8^n - 1$. Expressed formally, $P(n)$ is:

$$\exists y \in \mathbb{Z},\ 7 \cdot y = 8^n - 1.$$

We want to prove for all $n \in \mathbb{N}$ that $P(n)$ holds.

Base Case: $n = 0$. We want to prove that $P(0)$ is true.

We know that $8^0 - 1 = 0$, and that $7 \mid 0$. So $P(0)$ holds.

Inductive Step: Let $k \in \mathbb{N}$, and assume that $P(k)$ is true. That is, we assume that $7 \mid 8^k - 1$; unpacking the definition of divisibility, this means there exists y_k such that $8^k - 1 = 7y_k$.

Now we want to show that $P(k+1)$ holds:

$$7 \mid 8^{k+1} - 1, \text{ or in other words, } \exists y_{k+1} \in \mathbb{Z},\ 8^{k+1} - 1 = 7y_{k+1}.$$

How do we find this y_{k+1}? In order to prove $P(k+1)$ using $P(k)$, we have to extract the expression $8^k - 1$ out of the expression $8^{k+1} - 1$. Thus we will rewrite $8^{k+1} - 1$ as follows:

$$8^{k+1} - 1 = 8^{k+1} - 8 + 7 = 8(8^k - 1) + 7.$$

Next, we use the *induction hypothesis*, which says that $7y_k = 8^k - 1$:

$$\begin{aligned} 8^{k+1} - 1 &= 8(8^k - 1) + 7 \\ &= 8(7y_k) + 7 \\ &= 7(8y_k + 1) \end{aligned}$$

So let $y_{k+1} = 8y_k + 1$. Then $8^{k+1} - 1 = 7y_{k+1}$, and so $7 \mid 8^{k+1} - 1$. This completes the proof of the inductive step and thus the proof. □

Let's do another example, which is quite similar to the previous one, but is useful for practicing this new technique.

Example 3.3. For every natural number n, $n(n^2 + 5)$ is divisible by 6.

Proof. Let $P(n)$ be the statement that $n(n^2 + 5)$ is divisible by 6.

Base Case: $n = 0$.

When $n = 0$, the expression $n(n^2 + 5) = 0(0^2 + 5) = 0$. So it is divisible by 6 and thus $P(0)$ holds.

Inductive Step: Let $k \in \mathbb{N}$, and assume $P(k)$ is true. That is, we assume $k(k^2 + 5)$ is divisible by 6. We want to prove that $P(k + 1)$ holds; that is, we want to show that $(k + 1)((k + 1)^2 + 5)$ is divisible by 6.

As in the previous example, in order to prove $P(k+1)$ holds using the assumption that $P(k)$ holds, we somehow need to extract the expression $k(k^2 + 5)$ out of the expression $(k + 1)((k + 1)^2 + 5)$.

Some algebraic manipulations follow:

$$\begin{aligned} (k+1)((k+1)^2 + 5) &= (k+1)(k^2 + 2k + 6) \\ &= (k+1)\big((k^2 + 5) + (2k + 1)\big) \\ &= k(k^2 + 5) + k(2k + 1) + (k^2 + 5) + (2k + 1) \\ &= n(k^2 + 5) + 3k(k + 1) + 6 \qquad\qquad = k(k^2 + 5) + 3k^2 + 3k + 6 \end{aligned}$$

By the induction hypothesis, the first term on the right-hand side, $n(n^2 + 5)$, is a multiple of 6. For the second term, since n and $n + 1$ are consecutive natural numbers, one of them is even and thus $n(n + 1)$ is a multiple of 2 and thus $3n(n + 1)$ is a multiple of 6. Since each term on the right-hand side is a multiple of 6, their sum is also a multiple of 6, which completes the inductive step. □

Example 3.4. Now let us go back to our motivating example and prove it using induction. Recall that it was the statement that for any $m, x, y \in \mathbb{N}$ and for any $n \in \mathbb{N}$ that if $x \equiv y \pmod{m}$, then $x^n \equiv y^n \pmod{m}$.

Translation. Expressed formally we have:

$$\forall m, x, y \in \mathbb{N},\ \forall n \in \mathbb{N},\ x \equiv y \pmod{m} \Rightarrow x^n \equiv y^n \pmod{m}.$$

We have deliberately separated the three variables m, x, y from n, for a reason we'll discuss in next. *Discussion*. In the informal argument given at the start of the chapter, we first fixed values for m, x, y and then proved the claim when $n = 2$. Then for these same values of m, x, y we proved it for $n = 3$ and so on. In order to formalize this, we will want to first fix $m, x, y \in \mathbb{N}$ once and for all, and then prove the statement by induction on n.

Proof. Let $m, x, y \in \mathbb{N}$. Let $P(n)$ be the statement $x \equiv y \pmod{m} \Rightarrow x^n \equiv y^n \pmod{m}$. We want to prove that $\forall n \in \mathbb{N},\ P(n)$ by induction.

Note that this statement only makes sense after we have fixed m, x, and y.

Base Case: $n = 0$.

To prove this, we simply observe that when $n = 0$, the conclusion of the implication says that $x^0 \equiv y^0 \pmod{m}$, which is trivially true because both sides equal 1. So then this statement holds.

We didn't even need to use the assumption that $x \equiv y \pmod{m}$.

Inductive Step: Let $k \in \mathbb{N}$, and assume that $P(k)$ is true. That is, we assume that

$$P(k) : x \equiv y \pmod{m} \Rightarrow x^k \equiv y^k \pmod{m}.$$

From this assumption we want to prove

$$P(k+1) : x \equiv y \pmod{m} \Rightarrow x^{k+1} \equiv y^{k+1} \pmod{m}.$$

In order to prove $P(k+1)$.

Note that $P(k+1)$ has the form of an implication, so we know how we should proceed: assume the hypothesis, i.e., that $x \equiv y \pmod{m}$. Using our assumption that $P(k)$ is true, and that $x \equiv y \pmod{m}$, we can conclude that $x^k \equiv y^k \pmod{m}$.

In terms of proof structure: we assumed $A \Rightarrow B$, and then we assumed A. This allows us to conclude that B is true.

We know (from Example 2.15) that

$$(x^k \equiv y^k \pmod{m}) \wedge (x \equiv y \pmod{m}) \Rightarrow (x \cdot x^k \equiv y \cdot y^k \pmod{m}).$$

Since the left-hand side of this implication is true, the right hand side must also be true. Therefore $x^{k+1} \equiv y^{k+1} \pmod{m}$, and this completes the proof. □

One interesting subtlety in how we set up this proof is in how we chose the order of the variables m, x, y, n being quantified. You know already that changing the order of these variables doesn't change the meaning of the statement, because they are all universally-quantified.

However, changing their order *does* change the proof that we would write!

A different way to proceed in this proof would be to write the statement as

$$\forall n \in \mathbb{N},\ \forall m, x, y \in \mathbb{N},\ x \equiv y \pmod{m} \Rightarrow x^n \equiv y^n \pmod{m}.$$

Doing it this way, we would define $P(n)$ to be the (more complex) statement

$$\forall m, x, y \in \mathbb{N},\ x \equiv y \pmod{m} \Rightarrow x^n \equiv y^n \pmod{m}.$$

If we had proceeded this way, then the base case, $P(0)$ of the induction would be prove the implication for *all* values of m, x, y when $n = 0$. So in the base case we would first fix particular but arbitrary values of $m, x, y \in \mathbb{N}$ before proceeding with the proof. And again in the inductive step, we would need to prove $P(n)$ implies $P(n+1)$, which is a more complicated statement since the other variables m, x, y are not fixed but are universally quantified. When we have a universal statement such as this one that involves one universally quantified variable that we want to do induction on (in this case n), plus other universally quantified variables that we do not need to do induction on (in this case m, x, y), it is usually easier to first fix m, x, y and then do induction on n, as we did above, rather than the other way around.

We will do one more example from number theory. This example is proving an *inequality* rather than an equality, and demonstrates how to use induction with a different starting number as the base case.

Example 3.5. Prove that for all natural numbers n greater than or equal to 3, $2n + 1 \leq 2^n$.

Translation. We do the usual thing and express the "greater than or equal to 3" as a hypothesis in an implication.

$$\forall n \in \mathbb{N},\ n \geq 3 \Rightarrow 2n + 1 \leq 2^n.$$

This statement doesn't have exactly the right form for the induction technique we've learned, but if we define the predicate

$$P(n) : 2n + 1 \leq 2^n, \text{ where } n \text{ is a natural number}$$

then the statement becomes $\forall n \in \mathbb{N},\ n \geq 3 \Rightarrow P(n)$, which is close. *Discussion*. The principle of induction relies on two things: a base case, which gives us a starting point, and the induction step, which allows us to build on the base case to conclude the truth of the predicate for larger and larger natural numbers.

The particular number for the base case turns out not to be so important: if we prove that $P(3)$ is true as our base, then the induction step still allows us to conclude that $P(4), P(5), \ldots$ are all true!

Proof. **Base Case**: $n = 3$. Plugging in $n = 3$ into the left and right sides of the inequality, we get $7 \leq 8$, which is true.

Inductive Step: Let k be a natural number greater than or equal to 3, and assume that $P(k)$ is true: $2k + 1 \leq 2^k$. We want to prove $P(k+1)$ is true: $2(k+1) + 1 \leq 2^{k+1}$.

As usual, to obtain this inequality we start with the one we get from the induction hypothesis:

$$\begin{aligned} 2k + 1 &\leq 2^k \\ 2k + 1 + 2 &\leq 2^k + 2^k && \text{(since } 2 \leq 2^k\text{)} \\ 2(k+1) + 1 &\leq 2^{k+1} \end{aligned}$$

□

Exercise Break!

Use induction to prove each of the following statements.

3.1 For all $n \geq 1$, $9^n - 1$ is divisible by 8.

3.2 5^{2n-1} is divisible by 6.

3.3 $x^n - y^n$ is divisible by $x - y$.

3.4 For every $n \geq 1$, 2^{2^n} is divisible by at least n distinct primes.

3.5 For all $n \in \mathbb{N}$, $n \geq 6$, $5n + 5 \leq n^2$.

Combinatorics

Combinatorics is an area of mathematics concerned with counting objects, and more generally with analyzing patterns. A pattern is most typically a sequence of numbers and we will often want to derive a closed-form expression for a_k, the k^{th} number in the sequence, or for $\sum_{i=0}^{k} a_i$, the sum of the first $k+1$ numbers in the sequence.

We will start with a famous example. Consider the following sequence of numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots.$$

Call the k^{th} element in the sequence a_k. For each k, what is a_k? It isn't too hard to see that we obtain a_k by summing together the two previous numbers. That is, for all $k \geq 2$, $a_k = a_{k-1} + a_{k-2}$. This is a very famous sequence called the *Fibonacci sequence*.

Drawing inspiration from programming, our indexing starts at 0, not 1.

Another easier example is an *arithmetic sequence*. Suppose you start with 10 dollars, and every month you earn 200 dollars. How much money do you have after k months? At the start you have 10 dollars; after one month you have 210 dollars; after two months you have 410 dollars, etc. In general this gives rise to the sequence:

$$a_0 = 10, a_1 = 210, a_2 = 410, a_3 = 610, a_4 = 810, \ldots.$$

In general, $a_k = 10 + 200 \times k$. This is called an *arithmetic* sequence since we add a fixed amount each time.

Another kind of sequence is obtained by multiplying the current amount by a fixed value each time. Suppose that now you start with ten dollars, but now you invest your money in a very lucrative place so that every month your money doubles. This gives rise to the sequence:

$$a_0 = 10, a_1 = 20, a_2 = 40, a_3 = 80, a_4 = 160, \ldots.$$

It is not hard to see that in general, $a_k = 10 \times 2^k$. This is called a *geometric* sequence since we multiply by a fixed amount each time.

Finally, one more example. Let $n \in \mathbb{N}$. Suppose that we want to sum all natural numbers starting at 0, up to and including n. That is, $a_k = 0 + 1 + 2 + \cdots + k$. This gives rise to the infinite sequence:

$$a_0 = 0, a_1 = 1, a_2 = 3, a_3 = 6, a_4 = 10, a_5 = 15, \ldots.$$

It turns out that we have the following closed-form expression for a_n: $a_n = n \times (n+1)/2$.

In general, a sequence can be viewed as an ordered list of numbers given by the *range* of a function $f : \mathbb{N} \to \mathbb{R}$, where $a_0 = f(0)$, $a_1 = f(1)$, etc. The sequences we will study are infinite: there is one term a_k for each natural number k. We call the function f an *explicit* or *closed-form* expression/formula for the sequence. For example, the following is a closed-form expression for the Fibonacci sequence, known as Binet's formula:

$$a_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}.$$

Nice sequences will have explicit formulas, but there are also examples of sequences that are complex and that do not have an explicit formula. We can often use induction in order to prove that a particular explicit formula computes the terms in a sequence. Let's see some examples of this.

Example 3.6. Use induction to prove that the sum of the first n positive integers is equal to $n(n+1)/2$.

Translation. This statement can be translated as

$$\forall n \in \mathbb{N}, \sum_{j=1}^{n} j = n(n+1)/2.$$

Proof. Let $P(i)$ be the claim $\sum_{j=1}^{i} j = i(i+1)/2$.

Base Case:: $i = 0$. In this case, the left side is the empty sum (which has value o), and the right side is $0(0+1)/2 = 0$.

Inductive Step: Let $i \in \mathbb{N}$ and assume that $P(i)$ is true, i.e., that $\sum_{j=1}^{i} j = i(i+1)/2$. It is helpful to write down what we want to prove which is $P(i+1)$:

$$P(i+1) : \sum_{j=1}^{i+1} j = \frac{(i+1)(i+2)}{2}.$$

Now we have:

$$\begin{aligned}
\sum_{j=1}^{i+1} j &= \sum_{j=1}^{i} j + (i+1) \\
&= \frac{i(i+1)}{2} + (i+1) && \text{(by inductive hypothesis)} \\
&= \frac{i(i+1) + 2(i+1)}{2} \\
&= (i+1)(i+2)/2
\end{aligned}$$

This completes the proof. □

Example 3.7. The sum of the first n odd numbers is a perfect square.

Translation. This translates to the mathematical statement

$$\forall n \in \mathbb{N},\ \exists x \in \mathbb{N},\ \sum_{i=0}^{n-1} (2i+1) = x^2.$$

Discussion. We will try to prove this by induction on n. Let $P(n)$ be the statement that the sum of the first n odd numbers is a perfect square: $\exists x \in \mathbb{N} \sum_{i=0}^{n-1} (2i+1) = x^2$. The base case is $n = 1$. In this case we have $\sum_{i=0}^{0} (0+1) = 1 = 1^2$ so $P(1)$ is true. For the inductive step, we will assume $P(n)$ and try to prove $P(n+1)$:

$$\exists x_{n+1} \in \mathbb{N},\ \sum_{i=1}^{(n+1)-1} (2i+1) = x_{n+1}^2.$$

From the inductive hypothesis we know that the sum of the first n terms in the above sum is a perfect square. But how can we use that to deduce that when we add the last term, $2n+1$ to this perfect square that we will get yet another perfect square?

In order to get the proof to work we need to assume more. So let us look at some examples and try to learn more. We already saw that when $n = 1$, the sum of just this one odd number is a perfect square, 1^2. For $n = 2$ we have $1 + 3 = 4 = 2^2$. Thus the sum of the first two odd numbers is equal to 2^2. For $n = 3$ we have $1 + 3 + 5 = 9 = 3^2$. Now we start to see a pattern and we will conjecture that the sum of the first n odd numbers is equal to n^2. We will try to prove this stronger statement instead!

Proof. Let $P(n)$ be this new statement, expressed mathematically as:

$$\sum_{i=0}^{n-1}(2i+1) = n^2.$$

We will prove $P(n)$ by induction on n. As discussed above, $P(1)$ holds. For the inductive step, let us assume that $P(n)$ holds and try to prove $P(n+1)$. From the inductive hypothesis we now know that not only is the sum of the first n terms a perfect square, but it is n^2 so we have:

$$\sum_{i=0}^{(n+1)-1}(2i+1) = \sum_{i=0}^{n-1}(2i+1) + (2n+1) = n^2 + (2n+1) = (n+1)^2.$$

In the above, the first equality holds by algebraic manipulation; the second equality holds from the inductive hypothesis, and the third equality holds by algebraic manipulation. □

This next example is somewhat different in that we will want to prove something about objects that are not simply numbers.

Example 3.8. Prove that for every finite set S, $|\mathcal{P}(S)| = 2^{|S|}$.

Recall that $\mathcal{P}(S)$ is the *power set* of S, the set of all subsets of S. This statement is saying that if S has n elements, it has exactly 2^n subsets.

Translation. It may not be obvious how induction fits into this example, given that we are looking to prove something about sets, not natural numbers. There is, however, a nice approach we can take: perform induction using a variable representing the *size* of the set (note that the size of a finite set is always a natural number).

We say that we're performing induction *on* the size of the set.

Our inductive predicate is

$P(n)$: "every set S of size n satisfies $|\mathcal{P}(S)| = 2^n$," where n is a natural number.

The original statement is then equivalent to $\forall n \in \mathbb{N}, P(n)$, and we can use induction!

Proof. **Base Case**: $n = 0$. In this case, there is only one set of size 0: S must be the empty set. The only subset of the empty set is the empty set itself, so $\mathcal{P}(S) = \{\emptyset\}$ (size 1), and $2^0 = 1$.

Inductive Step: Now assume that $P(k)$ holds, and we want to prove $P(k+1)$. Note that the predicate P is really a universally-quantified statement ("every set S") with a condition ("of size k") Let S be a set, and assume S has size $k+1$. Let the elements of S be denoted by $s_1, \ldots, s_{k+1}$. We want to prove that the number of subsets of S is exactly 2^{k+1}.

First, consider all subsets of S that do not contain the last element, s_{k+1}; in other words, the subsets of $\{s_1, \ldots, s_k\}$. By the induction hypothesis, the number of such subsets is exactly 2^k.

Now consider all subsets of S that contain s_{k+1}. Again, the number of subsets of S that contain s_{k+1} is 2^k, since we can obtain these subsets by taking all 2^k subsets of $\{s_1, \ldots, s_k\}$, and adding s_{k+1} to each subset.

Thus in total there are $2^k + 2^k = 2^{k+1}$ subsets of S. □

Here's another example – the size of a set obtained as the Cartesian product of two finite sets. Try to prove it as an exercise; note that while there are two natural number variables here (n and m), you only need to do induction on one of them (and you can pick).

Example 3.9. Let A be a set size of n, and let B be a set of size m. Then $|A \times B| = n \cdot m$.

Recall that $A \times B$ is the set of all pairs (a, b) where $a \in A$ and $b \in B$.

Exercise Break!

3.6 Prove for all natural numbers i, $\displaystyle\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$.

3.7 Prove $\displaystyle\sum_{k=1}^{n} 5 \cdot 5^{k-1} = 5(2^n - 1)$.

3.8 (Handshake Theorem). Suppose you are at a party and n people (including yourself). At the end of the party, define a person's parity as Odd if they have shaken hands with an odd number of people, and Even if they have shaken hands with an even number of people. Prove that for all n, the number of people of Odd parity must be even.

Incorrect proofs by induction

Just as it is important to be able to formulate a correct proof by induction, it is equally important to not be fooled by an incorrect proof!

Consider this well-known example. Say we want to prove that all jellybeans have the same colour. Let $P(i)$ be the statement that any set of i jellybeans all have the same colour. The base case is when there is only one jellybean, and it has one colour, so the statement $P(1)$ is true.

Now let's assume that $P(i)$ is true and try to prove that $P(i+1)$ is true. Let $S = \{j_1, j_2, \ldots, j_{i+1}\}$ be a set of $(i+1)$ jellybeans. Consider the first i jellybeans in S: $S_1 = \{j_1, \ldots, j_i\}$. By the inductive hypothesis, they all must have the same colour. Now consider the last i jellybeans in S: $S_2 = \{j_2, \ldots, j_{i+1}\}$. Again by the inductive hypothesis, they must also have the same colour. Now since these two sets overlap, the two colours must be the same, thus the entire set $j_1, \ldots, j_{i+1}$ of jellybeans has the same colour and we can conclude $P(i+1)$.

We know that it is clearly wrong, so where exactly is the mistake? To find the error it is helpful to walk through a specific counterexample – say for instance we have two jellybeans, where the first one is red and the second one is yellow. In this case we can see the mistake since the two sets S_1 and S_2 do *not* overlap.

Looking ahead: strong induction

The way that we expressed the induction principle above was to prove the base case $P(0)$, and then give a general argument for $P(n+1)$ assuming $P(n)$. We said intuitively that this works by the domino effect: (1) Suppose we know that the first domino $P(0)$ is down, and (2) we know that as long as $P(n)$ is down, then so is $P(n+1)$, then this implies (3) that all of the dominoes are down. However, we could have replaced (2) by (2′) which states that as long as *all* of the first n dominoes are down, $P(0), \ldots, P(n)$, then so is $P(n+1)$. As long as we know (1) and (2′), this still implies (3). Note that proving (2′) rather than (2) may be *easier* since we can assume not only that $P(n)$ is true, but that all of $P(0), P(1), \ldots, P(n)$ are true, in order to deduce that $P(n+1)$ is true.

This is called the principle of *strong* induction. It turns out that strong induction and simple induction (the form we've been using in this chapter) are equivalent, but sometimes it can be easier to prove a statement using strong induction rather than simple induction.

More formally, suppose that we want to prove $\forall n \in \mathbb{N},\ n \geq k \Rightarrow P(n)$, where k is some natural number. The principle of strong induction can be used to prove this statement as follows.

- First, prove the base case $P(k)$.
- Secondly, prove that for any fixed but arbitrary $n \geq k$, $P(j)$ for all j,

$k \leq j \leq n$ implies $P(n+1)$.

Then we can conclude $\forall n \in \mathbb{N},\ n \geq k \Rightarrow P(n)$.

Example 3.10. Every integer $n \geq 2$ can be expressed as a product of one or more prime numbers.

Recall that a prime number is an integer greater than or equal to 2 that is only divisible by itself and 1. (The number 1 is not considered to be a prime number.)

Proof. Let $P(n)$ be the statement that n can be expressed as a product of one or more prime numbers. The base case is when $n = 2$. Since 2 is prime, 2 can be expressed as a product of one prime number (itself), and thus $P(2)$ is true.

For the inductive step, let n be an integer, $n \geq 2$. And assume that for every integer j, $2 \leq j \leq n$, that j can be expressed as a product of one or more prime numbers. Now we want to prove $P(n+1)$, that $n+1$ can also be expressed as a product of prime numbers. There are two cases. Either the integer $n+1$ is itself a prime number or it is not. If it is a prime number, then it is a product of one prime number (itself), and this case is complete. The second case is when $n+1$ is not a prime number, and thus $n+1 = a \cdot b$, where both a and b are positive integers that are both different from $n+1$ and 1. Since $2 \leq a \leq n$, and $2 \leq b \leq n$, by the induction hypothesis, both a and b can be written as the product of prime numbers, and thus $a \cdot b$ can also be written as the product of prime numbers and the proof is complete! □

Note that in this last example, it would have been futile to try to use simple induction since then we would only know that n is a product of prime numbers, which is useless in order to show that $n+1$ is the product of prime numbers.

Since whenever $n \geq 2$, n is not a factor of $n+1$.

4 Analysing Algorithm Running Time

When we first begin writing programs, we are mainly concerned with their correctness: do they work the way they're supposed to? As our programs get larger and more complex, we add in a second consideration: are they designed and documented clearly enough so that another person can read the code and make sense of what's going on? These two properties – correctness and design – are fundamental to writing good software. However, when designing software that is meant to be used on a large scale, or must react nearly instantaneously to a rapidly changing environment, there is a third consideration which must be taken into account when evaluating programs: the amount of time the program takes to run.

In this chapter, you will learn how to formally analyse the running time of an algorithm, and explain what factors do and do not matter when performing this analysis. You will learn the notation used by computer scientists to represent running time, and distinguish between best-, worst-, and average-case algorithm running times.

A motivating example

Consider the following function, which prints out all the items in a list:

```
1 def print_items(lst):
2     for item in lst:
3         print(item)
```

What can we say about the running time of this function? An empirical approach would be to measure the time it takes for this function to run on a bunch of different inputs, and then take the average of these times to come up with some sort of estimate of the "average" running time.

But of course, given that this algorithm performs an action for every

item in the input list, we expect it to take longer on longer lists, so taking an average of a bunch of running times loses important information about the inputs.

This is kind of like doing a random poll of how many birthday cakes a person has had, without taking into account how old that person is.

How about choosing one particular input, and calling the function multiple times on that input, and averaging those running times? This seems better, but even here there are some problems. For one, the computer's hardware can affect running time; for another, computers all are running multiple programs at the same time, and so what else is currently running on your computer also affects running time. So even running this experiment on one computer wouldn't necessarily be indicative of how long the function would take on a different computer, nor even how long it would take on the same computer but with a different number of other programs running.

While these sorts of timing experiments are actually done in practice for evaluating particular hardware or extremely low-level (close to hardware) programs, these details are often not helpful for the average software developer. After all, most software developers do not have control over the machine on which their software will be run.

So rather than use an empirical measurement of runtime, what we do instead is use an abstract representation of runtime: the number of "basic operations" an algorithm takes. However, there is a good reason "basic operation" is in quotation marks – this vague term raises a whole slew of questions:

- What counts as a "basic operation"?
- How do we tell which "basic operations" are used by an algorithm?
- Do all "basic operations" take the same amount of time?

The answers to these questions can depend on the hardware being used, as well as what programming language the algorithm is written in. Of course, these are precisely the details we wish to avoid thinking about.

For example, suppose we analysed the running time of the `print_items` function, counting only the `print` calls as basic operations. Then for a list of length n, there are n `print` calls, so we would say that the running time of `print_items` on a list of length n is n basic operations.

But then a friend comes along, and says "No wait, the variable `item` must be assigned a new value of the list at every loop iteration, and that counts as a basic operation." Okay, so then we would say that there are n print calls and n assignments to `item`, for a total running time of $2n$ basic operations for an input list of length n.

But then another friend chimes in, saying "But print calls take longer than variable assignments, since they need to change pixels on your monitor, so you should count each print call as 10 basic operations." Okay, so then there are n print calls worth $10n$ basic operations, plus the assignments to `item`, for a total of $11n$ basic operations for an input list of length n.

And then another friend joins in: "But you need to factor in an overhead of calling the function as a first step before the body executes, which counts as 1.5 basic operations (slower than assignment, faster than `print`)." So then we now have a running time of $11n + 1.5$ basic operations for an input list of length n.

And then another friend starts to speak, but you cut them off and say "That's it! This is getting way too complicated. I'm going back to timing experiments, which may be inaccurate but at least I won't have to listen to these increasing levels of fussiness."

The expressions n, $2n$, $11n$, and $11n + 1.5$ may be different mathematically, but they share a common *qualitative* type of growth: they are all lines, i.e., grow linearly with respect to n. What we will study in the next section is how to make this observation precise, and thus avoid the tedium of trying to exactly quantify our "basic operations," and instead measure the overall rate of growth of the number of them.

Asymptotic growth

Here is a quick reminder about function notation. When we write $f : A \to B$, we say that f is a function which maps elements of A to elements of B. In this chapter, we will mainly be concerned about functions mapping the natural numbers to the nonnegative real numbers, i.e., functions $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$. Though there are many different properties of functions that mathematicians study, we are only going to look at one such property: describing the long-term (asymptotic) growth of a function. We will proceed by building up a few different definitions of comparing function growth, which will eventually lead into one which is robust enough to be used in pratice.

These are the domain and range which arise in algorithm analysis – an algorithm can't take "negative" time to run, after all.

Definition 4.1. Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g is **absolutely dominated by** f if and only if for all $n \in \mathbb{N}$, $g(n) \leq f(n)$.

Example 4.1. Let $f(n) = n^2$ and $g(n) = n$. Then g is absolutely dominated by f.

Translation. This is a straightforward unpacking of a definition, which you should be very comfortable with by now: $\forall n \in \mathbb{N},\ g(n) \leq f(n)$.

Note that we aren't quantifying over f and g; the "let" in the example defines concrete functions that we want to prove something about.

Proof. Let $n \in \mathbb{N}$. We want to show that $n \leq n^2$.

Case 1: $n = 0$. In this case, $n^2 = n = 0$, so the inequality holds.

Case 2: $n \geq 1$. In this case, we take the inequality $n \geq 1$ and multiply both sides by n to get $n^2 \geq n$, or equivalently $n \leq n^2$. □

The direction of the inequality is preserved because we're multiplying by a non-negative number.

Unfortunately, absolute dominance is too strict for our purposes: if $g(n) \leq f(n)$ for every natural number except 5, then we can't say that g is absolutely dominated by f. For example, the function $g(n) = 2n$ is not absolutely dominated by $f(n) = n^2$, even though $g(n) \leq f(n)$ everywhere except $n = 1$. Here is another definition which is a bit more flexible than absolute dominance.

Definition 4.2. Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is dominated by** f **up to a constant factor** if and only if there exists a positive real number c such that for all $n \in \mathbb{N}$, $g(n) \leq c \cdot f(n)$.

Example 4.2. Let $f(n) = n^2$ and $g(n) = 2n$. Then g is dominated by f up to a constant factor.

Translation. Once again, the translation is a simple unpacking of the previous definition.

$$\exists c \in \mathbb{R}^+,\ \forall n \in \mathbb{N},\ g(n) \leq cf(n).$$

Discussion. The term "constant factor" is revealing. We already saw that n is absolutely dominated by n^2, so if the n is multiplied by 2, then we should be able to multiply n^2 by 2 as well to get the calculation to work out.

The order of quantifiers matter! The choice of c is *not* allowed to depend on n.

Proof. Let $c = 2$, and let $n \in \mathbb{N}$. We want to prove that $g(n) \leq cf(n)$, or in other words, $2n \leq 2n^2$.

Case 1: $n = 0$. In this case, $2n^2 = 2n = 0$, so the inequality holds.

Case 2: $n \geq 1$. Taking the assumed inequality $n \geq 1$ and multiplying both sides by $2n$ yields $2n^2 \geq 2n$, or equivalently $2n \leq 2n^2$. □

Intuitively, "dominated by up to a constant factor" allows us to ignore multiplicative constants in our functions. This will be very useful in our running time analysis because it frees us from worrying about the exact constants used to represent numbers of basic operations: n, $2n$, and $11n$ are all *equivalent* in the sense that each one dominates the other two up to a constant factor.

However, this second definition is still a little too restrictive, as the inequality must hold for every value of n. Consider the functions $f(n) =$

n^2 and $g(n) = n + 90$. No matter how much we scale up f by multiplying it by a constant, $f(0)$ will always be less than $g(0)$, so we cannot say that g is dominated by f up to a constant. And again this is silly: it is certainly possible to find a constant c such that $g(n) \leq cf(n)$ for every value except $n = 0$. So we want some way of omitting the value $n = 0$ from consideration; this is precisely what our third definition gives us.

Definition 4.3. Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is eventually dominated by** f if and only if there exists $n_0 \in \mathbb{R}^+$ such that $\forall n \in \mathbb{N}$, if $n \geq n_0$ then $g(n) \leq f(n)$.

Example 4.3. Let $f(n) = n^2$ and $g(n) = n + 90$. Then g is eventually dominated by f.

Translation.

$$\exists n_0 \in \mathbb{R}^+,\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq f(n).$$

Discussion. Okay, so rather than finding a constant to scale up f, we need to argue that for "large enough" values of n, $n + 90 \leq n^2$. How do we know that value of n is "large enough?"

Since this is a quadratic inequality, it is actually possible to solve it directly using factoring or the quadratic formula. But that's not really the point of this example, so instead we'll take advantage of the fact that *we* get to choose the value of n_0 to pick one which is large enough.

Proof. Let $n_0 = 90$, let $n \in \mathbb{N}$, and assume $n \geq n_0$. We want to prove that $n + 90 \leq n^2$.

We will start with the left-hand side and obtain a chain of inequalities that lead to the right.

$$\begin{aligned} n + 90 &\leq n + n && \text{(since } n \geq 90\text{)} \\ &= 2n \\ &\leq n \cdot n && \text{(since } n > 2\text{)} \\ &= n^2 \end{aligned}$$

□

Intuitively, this definition allows us to ignore "small" values of n and focus on the long term, or asymptotic, behaviour of the function. This is particularly important for ignoring the influence of slow-growing terms in a function, which may affect the function values for "small" n, but eventually are overshadowed by the faster-growing terms. In the above

example, we knew that n^2 grows faster than n, but because an extra $+90$ was added to the latter function, it took a while for the faster growth rate of n^2 to "catch up" to $n + 90$.

Our final definition combines both of the previous ones, enabling us to ignore both *constant factors* and *small values of n* when comparing functions.

Definition 4.4 (Big-Oh)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is eventually dominated by** f **up to a constant factor**, if and only if there exist $c, n_0 \in \mathbb{R}^+$, such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $g(n) \leq cf(n)$.

In this case, we can also say that g **is Big-Oh of** f, and write $g \in \mathcal{O}(f)$.

We use $\in \mathcal{O}(f)$ here because we define $\mathcal{O}(f)$ to be the *set* of functions that are eventually dominated by f up to a constant factor:

$$\mathcal{O}(f) = \{g \mid g : \mathbb{N} \to \mathbb{R}^{\geq 0}, \exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow g(n) \leq cf(n)\}.$$

Example 4.4. Let $f(n) = n^2$ and $g(n) = 100n + 5000$. Then $g \in \mathcal{O}(f)$ (or in other words, $100n + 5000 \in \mathcal{O}(n^2)$).

Translation.

$$\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow 100n + 5000 \leq cn^2.$$

Discussion. The trick for this (and most Big-Oh questions) is to keep in mind that we have total freedom to choose the values of the constants c and n_0, based on the algebraic properties of the functions being compared. Keep the intuition in mind: c is a constant we can use to scale up the n^2, and n_0 is a constant we can use to make n large.

Proof. Let $c = 100$ and $n_0 = 50$. Let $n \in \mathbb{N}$, and assume that $n \geq n_0$. We want to show that $100n + 5000 \leq 100 \cdot n^2$.

There is certainly more than one possible pair of c and n_0 values which yield a correct proof.

We can obtain that inequality by starting with the assumption $n \geq 50$, and performing some standard manipulations:

$$\begin{aligned} 50 &\leq n \\ n + 50 &\leq 2n \\ n + 50 &\leq n^2 && \text{(since } n \geq 2\text{)} \\ 100n + 5000 &\leq 100n^2 \end{aligned}$$

□

Properties of Big-Oh

If we had you always write chains of inequalities to prove that one function is Big-Oh of another, that would get quite tedious rather quickly. Instead, in this section we will prove some properties of this definition which are extremely useful for combining functions together under this definition, which save you quite a lot of work in the long run.

Definition 4.5 (sum of functions)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We can define the **sum of** f **and** g as the function $f + g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ such that

$$\forall n \in \mathbb{N},\ (f+g)(n) = f(n) + g(n).$$

Theorem 4.1 (Big-Oh of sum)**.** *Let $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$. If $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$, then $f + g \in \mathcal{O}(h)$.*

Translation.

$$\forall f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0},\ \big(f \in \mathcal{O}(h) \wedge g \in \mathcal{O}(h)\big) \Rightarrow f + g \in \mathcal{O}(h).$$

Discussion. This is similar in spirit to the divisibility proofs we did in Chapter 2, which used a term (divisibility) that contained a quantifier. Here, we need to assume that f and g are both Big-Oh of h, and prove that $f + g$ is also Big-Oh of h.

Of course, here the definition of Big-Oh contains *three* quantifiers, but the idea is the same.

Assuming $f \in \mathcal{O}(h)$ tells us there exist positive real numbers c_1 and n_1 such that for all $n \in \mathbb{N}$, if $n \geq n_1$ then $f(n) \leq c_1 h(n)$. There similarly exist c_2 and n_2 such that $g(n) \leq c_2 \cdot h(n)$ when $n \geq n_2$. *Warning:* we can't assume that $c_1 = c_2$ or $n_1 = n_2$, or any other relationship between these two sets of variables.

We want to prove that there exist $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $f(n) + g(n) \leq ch(n)$.

The forms of the inequalities we can assume – $f(n) \leq c_1 h(n)$, $g(n) \leq c_2 h(n)$ – and the final inequality are identical, and in particular the left-hand side suggests that we just need to add the two given inequalities together to get the third. We just need to make sure that both given inequalities hold by choosing n_0 to be large enough, and let c be large enough to take into account both c_1 and c_2.

Proof. Let $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$, and assume $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$. By these assumptions, there exist $c_1, c_2, n_1, n_2 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$,

- if $n \geq n_1$, then $f(n) \leq c_1 h(n)$, and
- if $n \geq n_2$, then $g(n) \leq c_2 h(n)$.

We want to prove that $f + g \in \mathcal{O}(h)$, i.e., that there exist $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $f(n) + g(n) \leq ch(n)$.

Let $n_0 = \max\{n_1, n_2\}$ and $c = c_1 + c_2$. Let $n \in \mathbb{N}$, and assume that $n \geq n_0$. We now want to prove that $f(n) + g(n) \leq ch(n)$.

This ensures that $n_0 \geq n_1$ and $n_0 \geq n_2$.

Since $n_0 \geq n_1$ and $n_0 \geq n_2$, we know that (for this value of n),

$$f(n) \leq c_1 h(n)$$
$$g(n) \leq c_2 h(n)$$

Adding these two inequalities together yields

$$f(n) + g(n) \leq c_1 h(n) + c_2 h(n) = (c_1 + c_2)h(n) = ch(n).$$

□

One special case of Big-Oh: $\mathcal{O}(1)$

So far, we have seen Big-Oh expressions like $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$, where the function in parentheses has grown to infinity. However, not every function takes on larger and larger values as its input grows. Some functions are *bounded*, meaning they never take on a value larger than some fixed constant.

For example, consider the constant function $f(n) = 1$, which always outputs the value 1, regardless of the value of n. What would it mean to say that a function g is Big-Oh of this f? Let's unpack the definition of Big-Oh to find out.

$$g \in \mathcal{O}(f)$$
$$\exists c, n_0 \in \mathbb{R}^+,\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq cf(n)$$
$$\exists c, n_0 \in \mathbb{R}^+,\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq c \qquad \text{(since } f(n) = 1\text{)}$$

In other words, there exists a constant c such that $g(n)$ is eventually always less than or equal to c. We say that such functions g are **asymptotically constant** with respect to their input, and write $g = \mathcal{O}(1)$ to represent this.

Exercise Break!

4.1 Let $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, and let $y \in \mathbb{R}^+$ be an arbitrary real number.

Prove that if $f \in \mathcal{O}(y)$, then $f \in \mathcal{O}(1)$ (this is why we write $\mathcal{O}(1)$ and usually never see $\mathcal{O}(2)$ or $\mathcal{O}(165)$).

$f \in \mathcal{O}(y)$ here means "f is Big-Oh of the constant function $g(n) = y$."

Omega and Theta

Big-Oh is a useful way of describing the long-term growth behaviour of functions, but its definition is limited in that it is not required to be an exact description of growth. After all, the key inequality $g(n) \leq cf(n)$ can be satisfied even if f grows much, *much* faster than g. For example, we could say that $n + 10 \in \mathcal{O}(n^{100})$ according to our definition, but this is not necessarily informative.

In other words, the definition of Big-Oh allows us to express *upper bounds* on the growth of a function, but does not allow us to distinguish between an upper bound that is tight and one that vastly overestimates the rate of growth.

In this section, we will introduce the final new pieces of notation for this chapter, which allow us to express tight bounds on the growth of a function.

Definition 4.6 (Omega)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is Omega of** f if and only if there exist constants $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$, then $cf(n) \leq g(n)$. In this case, we can also write $g \in \Omega(f)$.

You can think of Omega as the inverse of Big-Oh: when $g \in \Omega(f)$, then f is a *lower* bound on the growth rate of g. For example, we can use the definition to prove that $n^2 - 5 \in \Omega(n)$.

We can now express a bound that is tight for a function's growth rate quite elegantly by combining Big-Oh and Omega: if f is both a lower and upper bound for g, then g must grow at the same rate as f.

Definition 4.7 (Theta)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is (Big-)Theta of** f if and only if g is both Big-Oh of f and Omega of f. In this case, we can write $g \in \Theta(f)$.

Most of the time, when people say "Big-Oh" they actually mean Theta, i.e., a Big-Oh upper bound is meant to be the tight one, because we rarely say upper bounds which overestimate the rate of growth if we can help it. However, in this course we will always use Θ when we mean tight bounds, because we will see some cases where coming up with tight bounds isn't always easy.

Equivalently, g is Theta of f if and only if there exist constants $c_1, c_2, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $c_1 f(n) \leq g(n) \leq c_2 f(n)$.

Example 4.5. Let $f(n) = n^2$ and $g(n) = n + 10$. Then $g \in \mathcal{O}(f)$, but $g \notin \Theta(f)$. That is, f is an upper bound for the growth rate of g, but it is not a tight upper bound.

We have the following properties of Omega and Theta, which can be proved in an analogous fashion to the one for Big-Oh.

Theorem 4.2 (Omega of sum)**.** *Let* $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$*. If* $f \in \Omega(h)$ *and* $g \in \Omega(h)$*, then* $f + g \in \Omega(h)$*.*

Theorem 4.3 (Theta of sum)**.** *Let* $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$*. If* $f \in \Theta(h)$ *and* $g \in \Theta(h)$*, then* $f + g \in \Theta(h)$*.*

The other common context for sums with Theta are when we add two functions with different growth rates. The following theorem makes precise our intuition that the bigger one is more important.

Theorem 4.4 (Theta of sum, general)**.** *Let* $f_1, f_2, h_1, h_2 : \mathbb{N} \to \mathbb{R}^{\geq 0}$*. If* $f_1 \in \Theta(h_1)$*,* $f_2 \in \Theta(h_2)$*, and* $h_1 \in \mathcal{O}(h_2)$*, then* $f_1 + f_2 \in \Theta(h_2)$*.*

Back to algorithms

Let us return to our example at the beginning of the chapter:

```
1 def print_items(lst):
2     for item in lst:
3         print(item)
```

How can we use our definition of Big-Oh to help us analyse the running time of this algorithm? Remember that we have proposed expressions like n, $2n$, $11n$, $11n + 1.5$, where n is the length of the input list.

By using Big-Oh notation, we no longer need to worry about the constants involved, and so don't need to worry about whether a single call to `print` counts as one or ten "basic operations." Moreover, by focusing on the long-term growth, we can also ignore lower-order terms like the 1.5 in $11n + 1.5$.

Just as switching from measuring real time to counting "basic operations" allows us to ignore the computing environment in which the program runs, switching from an exact step count to Big-Oh notation allows us to ignore machine- and programming language-dependent constants involved in the execution of the code.

Having ignored all these external factors, our analysis will concentrate on how the **size of the input** influences the running time of a program, where we measure running time just using Big-Oh notation, and not exact expressions.

Warning: the "size" of the input to a program can mean different things depending on the type of input, or even depending on the pro-

gram itself. Whenever you perform a running time analysis, be sure to clearly state how you are measuring and representing input size.

Because constants don't matter, we will use a very coarse measure of "basic operation" to make our analysis as simple as possible. For our purposes, a basic operation (or step) is **any block of code whose running time does not depend on the size of the input**.

To belabour the point a little, this depends on how we define input size. For integers, we usually will assume they have a fixed size in memory (e.g., 32 bits), which is why arithmetic operations take constant time. But of course if we allow numbers to grow infinitely, this is no longer true, and performing arithmetic operations will no longer take constant time.

This includes all primitive language operations like most assignment statements, arithmetic calculations, and list and string indexing. The one major statement type which does not fit in this category is a function call – the running time of such statements depends on how long that particular function takes to run. We'll revisit this in more detail later.

Our first procedure for algorithm analysis follows four steps:

1. Identify the blocks of code which can be counted as a single basic operation, because they don't depend on the input size.
2. Identify any loops in the code, which cause basic operations to repeat. You'll need to figure out how many times those loops run, based on the size of the input.
3. Use your observations from the previous two steps to come up with an expression for the number of basic operations used in this algorithm.
4. Convert this number of basic operations into a Big-Theta expression.

Because Theta expressions depend only on the fastest-growing term in a sum, *and* ignores constants, we don't even need an exact, "correct" expression for the number of steps. This allows us to be rough with our analysis, but still get the correct Theta expression.

Example 4.6. Consider the function `print_items`. We define input size to be the *number of items of the input list*. Prove that the running time of `print_items` is $\Theta(n)$, where n is the length of the list.

Proof. For this algorithm, each iteration of the loop can be counted as a single operation, because nothing in it (including the call to `print`) depends on the size of the input list.

This is actually a little subtle. If we consider the size of individual list elements, it could be the case that some take a much longer time to print than others (imagine printing a string of one-thousand characters vs. the number 5). But by defining input size purely as the number of items, we are implicitly ignore the size of individual items. The running time of a call to `print` does *not* depend on the length of the input list.

So the running time depends on the number of loop iterations. Since this is a `for` loop over the `lst` argument, we know that the loop runs n times, where n is the length of `lst`.

Thus the total number of basic operations performed is n, and so the running time is $\Theta(n)$. □

It is quite possible to have nested loops in a function body, and analyse the running time in the same fashion. The simplest method of tackling such functions is to count the number of repeated basic operations in a loop starting with the *innermost* loop and working your way out.

Example 4.7. Consider the following function.

```
def print_sums(lst):
    for item1 in lst:
        for item2 in lst:
            print(item1 + item2)
```

Prove that the function `print_sums` runs in time $\Theta(n^2)$, where n is the length of `lst`. (We will assume input size for a list is always its length, unless something else is specified.)

Proof. Let n be the length of `lst`.

The inner loop (`for item2 in lst`) runs n times (once per item in `lst`), and each iteration is just a single basic operation.

But the entire inner loop is itself repeated, since it is inside another loop. The outer loop runs n times as well, and each of its iterations takes n operations.

So then the total number of basic operations is

$$\text{cost for the inner loop} \times \text{number of times inner loop is repeated} = n \times n = n^2.$$

So the running time of this algorithm is $\Theta(n^2)$. □

Students often make the mistake, however, that the number of nested loops should always be the exponent of n in the Big-Oh expression. However, things are not that simple, and in particular, not every loop takes n iterations.

E.g., two levels of nesting always becomes $\Theta(n^2)$.

Example 4.8. Consider the following function:

```
def f(lst):
    for item in lst:
        for i in range(10):
            print(item + i)
```

This function runs in time $\Theta(n)$. (Even though it has a nested list!)

Proof. Let n be the length of the input list `lst`. The inner loop repeats 10 times, and each iteration is again a single basic operation, for a total of 10 basic operations. The outer loop repeats n times, and each iteration takes 10 steps, for a total of $10n$ steps. This is $\Theta(n)$.

Note that this doesn't depend on n.

Alternative, better approach. The inner loop's running time doesn't depend on the number of items in the input list, so we can count it as a single basic operation.

The outer loop runs n times, and each iteration takes 1 step, for a total of n steps, which is $\Theta(n)$. □

When we are analysing the running time of two blocks of code executed in sequence (one after the other), we add together their individual running times. The sum theorems are particularly helpful here, as it tells us that we can simply compute Theta expressions for the blocks individually, and then combine them just by taking the fastest-growing one. Because Theta expressions are a simplification of exact mathematical function expressions, taking this approach is often easier and faster than trying to count an exact number steps for the entire function.

I.e., $\Theta(n^2)$ is simpler than $15n^2 + 0.001n + 165$.

Example 4.9. Consider the following function, which is a combination of two previous functions.

```
1 def combined(lst):
2     # Loop 1
3     for item in lst:
4         for i in range(10):
5             print(item + i)
6     # Loop 2
7     for item1 in lst:
8         for item2 in lst:
9             print(item1 + item2)
```

Proof. Let n be the length of `lst`. We have already seen that the first loop runs in time $\Theta(n)$, while the second loop runs in time $\Theta(n^2)$.

By "runs in time $\Theta(n)$," we mean that the number of basic operations of the second loop is a function $f(n) \in \Theta(n)$.

By Theorem 4.4, we can conclude that `combined` runs in time $\Theta(n^2)$. (Since $n \in \mathcal{O}(n^2)$.) □

Loop iterations with changing costs

Consider the following function:

```
def all_pairs(lst):
    i = 0
    while i < len(lst):
        j = 0
        while j < i:
            print(i + j)
            j = j + 1
        i = i + 1
```

Like previous examples, function has a nested loop. However, unlike those examples, here the inner loop's running time depends on the current value of `i`, i.e., which iteration of the outer loop we're on.

This means we cannot take the previous approach of calculating the cost of the inner loop, and multiplying it by the number of iterations of the outer loop; this only works if the cost of each outer loop iteration is the same.

So instead, we need to manually add up the cost of each iteration of the outer loop, which depends on the number of iterations of the inner loop. More specifically, since j goes from 0 to i, the number of iterations of the inner loop is i, and each iteration of the inner loop counts as one basic operation. So the cost of the i-th iteration of the outer loop is $i + 1$, where the 1 comes from counting the assignment statements in the outer loop.

So then to compute the total cost, we must add up the cost of each iteration. Let n be the length of the input list, and $T(n)$ be the running time of `all_pairs` on a list of length n.

$$T(n) = \sum_{i=1}^{n}(i+1) = \frac{n(n+1)}{2} + n \in \Theta(n^2).$$

Helper functions

Finally, let us return to how we deal with helper functions in our analysis. Suppose we are asked to analyse the running time of the following function under the assumption that the helper functions do not change the size of `lst`:

```
def uses_helpers(lst):
    x = helper1(lst)
    y = helper2(lst)
    return x + y
```

As with analysing any other sequential program, we simply take the sum of each individual code block's running time. That is, we take the running time of `helper1` when given input `lst`, the running time of `helper2` when given input `lst`, and the single basic operation for `return x + y`, and add these together. We do not need to add any "extra overhead" for calling functions: while this overhead often exists, it does not depend on the size of the input, and so we treat this as a single basic operation that can be ignored.

Any constant number of basic operations is dominated by terms which grow with the size of the input.

Example 4.10. If `helper1` runs in time $\Theta(n^2)$ and `helper2` runs in time $\Theta(n^3)$, where the n in both cases is the size of their input list, then `uses_helper` runs in time $\Theta(n^3)$.

Proof. Let n be the size of the input to `uses_helpers`. Then because `helper1` is called on the same input, it takes time $\Theta(n^2)$. Similarly, `helper2` takes time $\Theta(n^3)$. Finally, the cost of the `return` statement is $\Theta(1)$.

Taking the sum of these yields a total running time of $\Theta(n^3)$. □

Note that unlike previous examples, this analysis was an implication: the running time of `uses_helpers` depends on the running times of `helper1` and `helper2`. It is important to keep this in mind when both writing and analysing your code: it is easy to skim over a helper function call because it takes up so little visual space, but that one call might make the difference between a $\Theta(n)$ and $\Theta(2^n)$ running time.

Worst-case and best-case running times

In the previous section, we saw how to use asymptotic notation to characterize the *rate of growth* of the number of "basic operations" as a way of analysing the running time of an algorithm. This approach allows us to ignore details of the computing environment in which the algorithm is run, and machine- and language-dependent implementations of primitive operations, and instead characterize the relationship between the input size and number of basic operations performed.

However, this focus on just the input size is a little too restrictive. Even though we can define input size differently for each algorithm we analyse, we tend not to stray too far from the "natural" definitions (e.g., length of list). In practice, though, algorithms often depend on the actual value of the input, not just its size. For example, consider the following function, which searches for an even number in a list of integers.

```
def has_even(numbers):
    for number in numbers:
        if number % 2 == 0:
            return True
    return False
```

Because this function returns as soon as it finds an even number in the list, its running time is not necessarily proportional to the length of the input list.

The running time of a function can vary even when the input size is fixed. The question "what is *the* running time of `has_even` on an input of length n?" does not make sense, as depending on what positions in the list have even numbers, the algorithm could be as fast as $\Theta(1)$ (not depending on the length of the list at all), to as slow as $\Theta(n)$ time (having to check every single item in the list).

And because our asymptotic notation is used to describe the growth rate of *functions*, we cannot use it to describe the growth of a whole range of values with respect to increasing input sizes. A natural approach to fix this problem is to focus on the extremeties of this range of running times.

Definition 4.8 (worst-/best-case running time)**.** For a fixed algorithm `f`, we can define the following two functions:

$$WC_f(n) = \max\{\text{running time of executing } \texttt{f(x)} \mid \texttt{x} \text{ has size } n\}$$
$$BC_f(n) = \min\{\text{running time of executing } \texttt{f(x)} \mid \texttt{x} \text{ has size } n\}$$

Here, "running time" is measured in exact number of basic operations. We are taking the maximum/minimum of a set of numbers, *not* asymptotic expressions.

We call the function WC_f the **worst-case running time of** f and BC_f the **best-case running time of** f.

Note that WC_f and BC_f are functions, not (constant) numbers: they describe maximum and minimum possible running times for an input of size n, for every natural number n. And because they are functions, we can use Big-Oh notation to describe them, saying things like "the worst-case running time of this function is $\Theta(n^2)$, but its best-case running time is $\Theta(n)$."

However, it takes a bit more work to obtain tight bounds on these running times. Let's think about just the worst-case running time for now. It is difficult to compute the *exact maximum number of basic operations* performed by this algorithm for every input size, which requires

that we identify an input for each input size, count its maximum number of basic operations, and then prove that every input of this size takes at most this number of operations.

Instead, our (easier) first step is to skip the "identify an input" part, and simply find a number of steps (in terms of the input size) which is an *upper bound* on the number of basic operations for all possible inputs. Unlike our previous examples, finding this number of steps does not rely on tracing the function's behaviour on a given input, but instead analysing the properties of the code itself.

Example 4.11. For any $n \in \mathbb{N}$, the number of basic operations for `has_even` on any input list of length n is at most $n + 1$.

Translation. There is a key piece of the translation which reveals good insight into what we're trying to do here. The "any input list of length n" is an *universal quantification*: we are really proving a statement about all possible inputs to `has_even` (of length n). We can therefore express this statement as:

"For every n and every list `numbers` of length n, `has_even(numbers)` takes at most $n + 1$ basic operations."

Proof. We will let $n \in \mathbb{N}$, and let `numbers` be an arbitrary list of length n. We want to show that `has_even(numbers)` takes at most $n + 1$ basic operations.

Note that we can't assume anything about the values inside `numbers`. However, we can still make some observations about the code:

- The loop (`for number in numbers`) iterates *at most* n times. Each loop iteration counts as a single basic operation, so the loop takes at most n basic operations.
- The `return False` statement (if it is executed) counts as 1 basic operation.

The total number of basic operations possible is simply their sum: $n + 1$. □

Note that we did *not* prove that `has_even(numbers)` takes exactly $n + 1$ basic operations for an arbitrary input `numbers` (this is false); we only proved an *upper bound* on the number of operations. And in fact, we don't even care that much about the exact number: what we ultimately care about is the asymptotic growth rate, which is linear for $n + 1$. This

allows us to conclude that the worst-case running time of `has_even` is $\mathcal{O}(n)$, where n is the length of the input list. Note that we must use Big-Oh here, not Theta: we don't yet know that this upper bound is tight.

If this is surprising, note that we could have done the above proof but replaced $n + 1$ by $5000n + 165$ and it would still have been valid.

So how do we prove this is tight? Since we've just shown that $WC(n) \in \mathcal{O}(n)$, we need to prove the corresponding lower bound $WC(n) \in \Omega(n)$. But what does it mean to prove a lower bound on the maximum of a set of numbers? Suppose we have a set of numbers S, and say that "the maximum of S is at least 50." This doesn't tell us what the maximum of S actually is, but it does give us one piece of information: there has to be a number in S which is at least 50. The converse is also true – if I tell you that S contains the number 50, then you can conclude that the maximum of S is at least 50.

$$\max S \geq 50 \Leftrightarrow (\exists x \in S,\ x \geq 50).$$

So to find a lower bound on the worst-case running time, we need a set of inputs, one per input size, whose running time is "large" (i.e., close to the upper bound of $n + 1$). This motivates the following definition:

Definition 4.9 (worst-/best-case input family)**.** Let `f` be an algorithm. A **worst-case input family** is a set of inputs to `f`, one for each input size, whose running time is $\Theta(WC_f)$.

A **best-case input family** is a set of inputs to `f`, one for each input size, whose running time is $\Theta(BC_f)$.

Note that "worst-case input family" is not the same as "maximum running time input family". A family of inputs which take $n/2$ basic operations is still considered a worst-case input family, since its running time matches the asymptotic upper bound of $\mathcal{O}(n)$. This is the example we'll next look at, just to reinforce this idea.

Example 4.12. For each $n \in \mathbb{N}$, consider the list which has all 1's in the first $\lfloor n/2 \rfloor$ spots, and all 0's in the remaining spots. The running time of `has_even` on this family of inputs is $\Theta(n)$.

Proof. Now we have a concrete set of inputs, so we can use the same analysis we did in the previous sections. Let $n \in \mathbb{N}$, and let `numbers` be the input of length n from this family. That is, `numbers` has all 1's in the first $\lfloor n/2 \rfloor$ spots, and all 0's in the remaining spots.

Then the loop iterates exactly $\lfloor n/2 \rfloor + 1$ times, and since each loop iteration counts as one basic operation, this is the number of basic operations as well. So the running time of this family is $\Theta(n)$. □

To finish off our example, then, we take the upper bound from Example 4.11 and the input family from Example 4.12 to obtain a tight bound on the worst-case running time of `has_even`.

Example 4.13. The worst-case running time of `has_even` is $\Theta(n)$, where n is the length of the input list.

Proof. Let $WC(n)$ be the worst-case running time function of `has_even`. We have already seen that $WC(n) \in \mathcal{O}(n)$. We want to prove that $WC(n) \in \Omega(n)$.

Since we have a family of inputs that runs in time $\Theta(n)$, then $WC(n) \in \Omega(n)$ (i.e., the maximum running time is greater than or equal to the running time of this input family). □

To summarize, to obtain a tight bound on the worst-case running time of a function, we need to do two things:

- Use the properties of the code to obtain an *asymptotic upper bound* on the worst-case running time. We would say something like $WC_f(n) \in \mathcal{O}(g(n))$.
- Find a family of inputs whose running time is $\Theta(g(n))$. This will indirectly prove that $WC_f(n) \in \Omega(g(n))$, and so we can conclude that $WC_f(n) \in \Theta(g(n))$.

And prove this running time, of course.

There is an analogous process for the best-case running time, although this is less commonly studied:

- Use the properties of the code to obtain an *asymptotic lower bound* on the worst-case running time. We would say something like $BC_f(n) \in \Omega(g(n))$.
- Find a family of inputs whose running time is $\Theta(g(n))$. This will indirectly prove that $BC_f(n) \in \mathcal{O}(g(n))$, and so we can conclude that $BC_f(n) \in \Theta(g(n))$.

Note that the roles of upper/lower bounds have switched. If we have a set S and say that it contains the number 50, we can conclude that the *minimum* number in S is *at most* 50.

Don't assume bounds are tight!

It is likely unsatisfying to hear that upper and lower bounds really are distinct things that must be computed separately. Our intuition here pulls us towards the bounds being "obviously" the same, but this is really a side effect of the examples we have studied so far in this course being rather straightforward. But this won't always be the case: the study of more complex algorithms and data structures exhibits quite a

few cases where obtaining an upper bound involves a completely different argument from a lower bound.

Here's a taste: a brute-force algorithm that prints all the prime numbers less than or equal to an input number.

```
def print_primes(n):
    for k in range(2, n + 1):
        if not is_prime(k):
            print(k)


def is_prime(n):
    if n < 2:
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True
```

What is the asymptotic running time of `print_primes` as a function of n? It seems at first glance this should be straightforward to analyse, as we do not need to worry about worst-case here.

The problem lies in the `is_prime` helper. Because it stops as soon as it finds a factor of n between `2` and $n-1$, the number of iterations that occur can vary between `1` and $n-2$. Note that `is_prime` only goes through all $n-2$ iterations if n is prime.

So if we want to analyse the running time of `print_primes`, we need to add up the cost of running `is_prime` for each number between `2` and $n-1$. Let $T_1(n)$ represent the running time of `print_primes(n)`, and $T_2(n)$ represent the running time of `is_prime(n)`.

We can ignore the other constant-time operations in `print_primes` and `is_prime`.

$$T_1(n) = \sum_{k=2}^{n} T_2(k)$$

How do we evaluate this sum? We could say that the running time of `is_prime(k)` is at most $k-2$, but this forces us to change the equality into an inequality:

$$\begin{aligned} T_1(n) &\leq \sum_{k=2}^{n} (k-2) \\ &= \sum_{k=2}^{n} k - 2(n-1) \\ &= \sum_{k=1}^{n} k - 2(n-1) - 1 \\ &= \frac{n(n+1)}{2} - 2n + 1 \end{aligned}$$

In other words, we get a quadratic (n^2) running time here. But because our analysis over-estimated the running time of `is_prime(k)`, this is only an *upper bound* on the running time: $T_1(n) \in \mathcal{O}(n^2)$.

In fact, this analysis did not take into account `is_prime` stopping early at all! However, it is not at all obvious how to take this into account in our analysis, since we lack the mathematical tools required to think about when and how `is_prime` stops early for the different values of k.

However, here is one simple argument that we could use to get a *lower bound* on the running time of this function. We observed that `is_prime` runs for the maximum number of iterations when its input is prime, or in other words, $T_2(k) = k - 1$ when k is prime. So what do we get if we take the original expression for $T_1(k)$ and throw out all the terms except when k is prime?

$$\begin{aligned} T_1(n) &= \sum_{k=2}^{n} T_2(k) \\ &\geq \sum_{\substack{k \leq n \\ k \text{ is prime}}} T_2(k) \\ &= \sum_{\substack{k \leq n \\ k \text{ is prime}}} (k-2) \\ &= \sum_{\substack{k \leq n \\ k \text{ is prime}}} k - 2 \times \#\text{ of primes} \leq n \end{aligned}$$

We know from number theory that the sum of the primes $\leq n$ is roughly $\frac{n^2}{\log n}$, and the number of primes $\leq n$ is roughly $\frac{n}{\log n}$. This means that $T_1(n) \in \Omega\left(\frac{n^2}{\log n}\right)$.

Notice that this doesn't match our upper bound! Does that mean that

one of these is wrong? Not quite – it means that the true running time is somewhere between $\frac{n^2}{\log n}$ and n^2, but we need to perform a better analysis to find it out.

Average-case analysis

So far, we have only been concerned with the extremes of algorithm analysis. However, in practice this type of analysis often ends up being misleading, with a variety of algorithms and data structures having a poor worst-case performance still yet performing well on the vast majority of inputs.

Some reflection makes this not too surprising; focusing on the maximum of a set of numbers says very little about the "typical" number in that set, or, more precisely, nothing about the *distribution* of numbers in that set.

A bit more concretely, suppose we have an algorithm `f`, and we look at the set of running times

$$Times_{f,n} = \{\text{running time of executing } \texttt{f(x)} \mid \texttt{x} \text{ has size } n\}.$$

We have seen that we define the worst-case running time with the maximum running time in this set, and the best-case running time with the minimum. Our final topic of this chapter will be to look at another measure of the running time: taking the *average* of the numbers in this set.

Don't forget that the worst-/best-case running times are *functions* that use not just one set but all the $Times_{f,n}$ sets.

A first example

Consider the following algorithm, which searches for a particular item in a list.

```
1 def search(lst, x):
2     for item in lst:
3         if item == x:
4             return True
5     return False
```

Let n represent the length of `lst`. The loop body counts as one basic operation, and so the running time of this algorithm is proportional

on the number of loop iterations. The loop can iterate between 1 and n times, leading to an upper bound on the worst-case of $\mathcal{O}(n)$ and a lower bound on the best-case of $\Omega(1)$. We'll leave it as an exercise to show that these bounds are tight (this is basically the same analysis we did in the previous section). But what can we say about the average of all possible inputs of length n?

Well, for one thing, we need to precisely define what we mean by "all possible inputs of length n." Because we don't have any restrictions on the elements stored in the input list, it seems like there could be an infinite number of lists of length n to choose from, and we cannot take an average of an infinite set of numbers.

So let us focus on one particular set of allowable inputs. We define the set $\mathcal{I}_n$ of inputs to be pairs `(lst, 1)` where `lst` is any permutation of the numbers $\{1, 2, \ldots, n\}$, and we are always searching for the number `1` in the list.

This forces the item being searched for to always be in `lst`, so we might hope that the average running time is faster than the worst-case because of early returns.

Example 4.14. Given this set of inputs $\mathcal{I}_n$, prove that the average-case running time of `search` is $\Theta(n)$.

Proof. We first want to calculate an *exact* expression for

$$Avg_{search}(n) = \frac{1}{|\mathcal{I}_n|} \sum_{(lst,1)\in\mathcal{I}_n} \text{running time of } \texttt{search(lst, 1)}.$$

Note that $|\mathcal{I}_n| = n!$, since this is the number of permutations of $\{1, \ldots, n\}$.

$$Avg_{search}(n) = \frac{1}{n!} \sum_{(lst,1)\in\mathcal{I}_n} \text{running time of } \texttt{search(lst, 1)}.$$

Also, we want to make explicit that the inner summation ranges over values for `lst`, so we define S_n to be the set of all permutations of $\{1, \ldots, n\}$, and write

$$Avg_{search}(n) = \frac{1}{n!} \sum_{lst\in S_n} \text{running time of } \texttt{search(lst, 1)}.$$

Now, the running time of `search(lst, 1)` is the number of loop iterations performed, and this is exactly equal to the position that 1 appears in `lst` (assuming here that list indexing begins at 1, not 0).

So we can rewrite the sum as follows:

$$Avg_{search}(n) = \frac{1}{n!} \sum_{lst \in S_n} \text{position of 1 in } \texttt{lst}$$

Now, it might be challenging to compute this sum, since 1 could appear in any position in `lst`. However, we can *split up* S_n based on the position that 1 appears:

$$\begin{aligned} Avg_{search}(n) &= \frac{1}{n!} \sum_{i=1}^{n} \sum_{\substack{lst \in S_n \\ 1 \text{ is at } \texttt{lst[i]}}} \text{position of 1 in } \texttt{lst} \\ &= \frac{1}{n!} \sum_{i=1}^{n} \sum_{\substack{lst \in S_n \\ 1 \text{ is at } \texttt{lst[i]}}} i \end{aligned}$$

For the inner summation, we are not using `lst` in the summation, so it just adds up i a bunch of times. To figure out the number of times i is added together, we need to count the number of lists `lst` which have 1 at position i. There are $(n-1)!$ such lists: once we have fixed position i to be 1 in the list, the remaining spots can be any of the $(n-1)!$ permutations of $\{2, \ldots, n\}$. Using this allows us to obtain a final expression for $Avg_{search}(n)$:

$$\begin{aligned} Avg_{search}(n) &= \frac{1}{n!} \sum_{i=1}^{n} \sum_{\substack{lst \in S_n \\ 1 \text{ is at } \texttt{lst[i]}}} i \\ &= \frac{1}{n!} \sum_{i=1}^{n} i(n-1)! \\ &= \frac{1}{n} \sum_{i=1}^{n} i \\ &= \frac{1}{n} \cdot \frac{n(n+1)}{2} \\ &= \frac{n+1}{2} \end{aligned}$$

In other words, the average running time of search on this set of inputs is $\frac{n+1}{2} \in \Theta(n)$. □

Example 4.15. Now consider the set of inputs $\mathcal{I}'_n$, which contains all pairs `(lst, x)` where `lst` is a permutation of $\{1, \ldots, n\}$ and `x` is any number between 1 and n.

Note that x is still guaranteed to be in `lst`.

Proof. While we want to perform the basically same calculation:

$$Avg_{search}(n) = \frac{1}{|\mathcal{I}'_n|} \sum_{(lst,x) \in \mathcal{I}'_n} \text{running time of } \texttt{search(lst, x)}.$$

Note that this seems like a generalization of the previous set of inputs: we now have $|\mathcal{I}'_n| = n \cdot n!$, since now for each permutation we have n choices for `x`. However, we can do some manipulation of the sum to obtain the exact expression we computed in the previous example:

$$\begin{aligned}
Avg_{search}(n) &= \frac{1}{|\mathcal{I}'_n|} \sum_{(lst,x) \in \mathcal{I}'_n} \text{running time of } \texttt{search(lst, x)} \\
&= \frac{1}{n \cdot n!} \sum_{(lst,x) \in \mathcal{I}'_n} \text{running time of } \texttt{search(lst, x)} \\
&= \frac{1}{n \cdot n!} \sum_{x=1}^{n} \sum_{lst \in S_n} \text{running time of } \texttt{search(lst, x)} \\
&= \frac{1}{n} \sum_{x=1}^{n} \left(\frac{1}{n!} \sum_{lst \in S_n} \text{running time of } \texttt{search(lst, x)} \right)
\end{aligned}$$

We have done two main things: explicitly pulled out the summation over `x`, so now the part in parentheses has a fixed `x` value; we pulled in the constant $1/n!$, which makes the term in parentheses look exactly like our previous calculation, except with 1 replaced by `x`.

Why is this useful? Well, we already know that

$$\frac{1}{n!} \sum_{lst \in S_n} \text{running time of } \texttt{search(lst, 1)} = \frac{n+1}{2}.$$

But in our above proof, we didn't really use any special properties of 1 at all, other than the fact it was one of the numbers guaranteed to be in the list. So in fact, for *any* value of `x` between 1 and n, the same equality holds:

$$\frac{1}{n!} \sum_{lst \in S_n} \text{running time of } \texttt{search(lst, x)} = \frac{n+1}{2}.$$

This results in an absolutely massive simplification of our original expression:

$$\begin{aligned} Avg_{search}(n) &= \frac{1}{n}\sum_{x=1}^{n}\left(\frac{1}{n!}\sum_{lst \in S_n} \text{running time of } \texttt{search(lst, x)}\right) \\ &= \frac{1}{n}\sum_{x=1}^{n}\frac{n+1}{2} \\ &= \frac{n+1}{2} \end{aligned}$$

Given the symmetry for different possible x values, it is perhaps not too surprising that the exact step count is the same for the two examples. You would expect this to change, however, if we expanded the possible values of x to, say, $\{1, \ldots, 2n\}$.

This leads to an average-case running time of $\frac{n+1}{2}$ steps, which is $\Theta(n)$. □

Notice that we do not need to compute an upper and lower bound separately, since in this case we have computed an exact average. (Much like if we had the exact set of inputs, we can compute the exact max and exact min, and don't need to compute upper and lower bounds separately.)

Like worst-case and best-case running times, the average-case running time is a *function* which relates input size to some measure of program efficiency. In this particular example, we found that for the given set of inputs $\mathcal{I}_n$ for each n, the average-case running time is asymptotically equal to that of the worst-case.

This might sound a little disappointing, but keep in mind the positive information this tells us: the worst-case input family here is not so different from the average case, i.e., it is fairly representative of the algorithm's running time as a whole.

It is not always the case that the average-case running time is aymptotically the same as the worst-case running time. It is certainly possible for the average-case to the asymptotically the same as the best-case, or lie somewhere in between best- and worst-cases. It is also very sensitive to the set of inputs you choose to analyse, as you'll explore in the exercise. In CSC263, you will return to this idea of average-case input with more sophisticated examples, looking not just at more complex functions, but also introducing the notions of probability into the analysis, allowing different inputs to be chosen more frequently than others.

Exercise Break!

4.2 Consider this alternate set of inputs for SEARCH: $\mathcal{J}_n$, where for each input $(lst, x) \in \mathcal{J}_n$, lst has length n, and x and the elements of lst are all between the numbers 1 and 10 (of course, lst can now contain duplicates).

Show that the average-case running time of SEARCH on this set of inputs is $\Theta(1)$, i.e., is constant with respect to the length of the input list.

5 Graphs and Trees

Our final mathematical domain of study is a powerful and ubiquitous way of representing entities and the relationships between them. If this sounds generic, that's because it is: this type of representation is abstract enough that we can use it to model concepts as varied as geographic locations and routes to animals and plants in an ecosystem to people in a social network.

In this chapter, you will begin your study of *graph theory*, learning how to precisely define different types of these models (called graphs), and (of course) state and prove properties of these entities.

Initial definitions

Let us start with some basic definitions.

Definition 5.1 (graph, vertex, edge)**.** A **graph** is a pair of sets (V, E), which are defined as follows:

- V is a set of objects, where each element of V is called a **vertex** of the graph.
- a set E of pairs of objects, where each pair (v_1, v_2) consists of two distinct vertices – i.e., $v_1, v_2 \in V$ – and is called an **edge** of the graph.

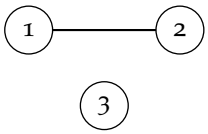
  Order does not matter in the pairs, and so (v_1, v_2) and (v_2, v_1) represent the same edge.

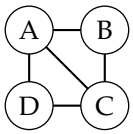
The conventional notation to introduce a graph is to write $G = (V, E)$, where G is the graph itself, V is its vertex set, and E is its edge set.

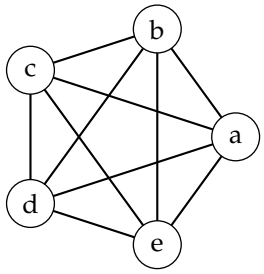
This is a concise way to introduce three variables in one expression.

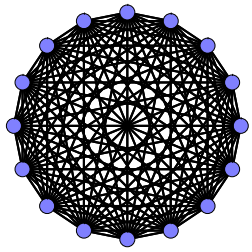
Intuitively, the set of vertices of a graph represents a collection of objects, and the set of edges of a graph represent the relationships between those objects. For example, if we wanted to use the terminology of graphs to describe Facebook, we could say that each Facebook user is a *vertex*, while each friendship between two Facebook users is an *edge* between the corresponding vertices.

We often draw graphs using dots to represent vertices, and line segments to represent edges. We have drawn some examples of graphs below.

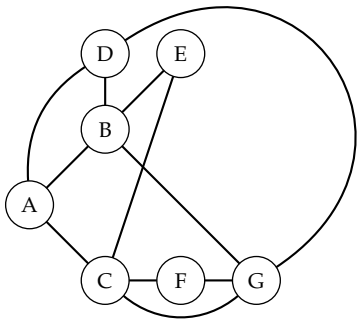








Example 5.1. Consider the graph on the right. How many vertices and how many edges does it have?



Discussion. This isn't a proof question, but just an exercise in terminology. To answer this, I have to be comfortable with the terminology *vertices* and *edges*, as well as pictoral representations of graphs. I just need to remember that dots correspond to vertices, and lines correspond to edges. (There are seven vertices and nine edges.)

Now that we have these definitions in hand, let us prove our first general graph property. Unlike the previous example, here we will not have a concrete graph to work with, but instead have to work with an arbitrary graph.

Hearing this, you should immediately expect to see a universal quantification over the set of all possible graphs.

Example 5.2. Let $G = (V, E)$ be an arbitrary graph. Then $|E| \leq \dfrac{|V|(|V|-1)}{2}$.

Translation. Because our proof introduces G as an arbitrary graph, we know that the statement we are proving is universally-quantified statement over the set of all possible graphs. We will denote this set $\mathcal{G}$:

$$\forall G = (V, E) \in \mathcal{G},\ |E| \leq \frac{|V|(|V|-1)}{2}.$$

it's a $\forall x \in D,\ P(x)$

Note that the *structure* of the statement is pretty straightforward, with the only tricky bit being that G is not an arbitrary *number*, but an arbitrary *graph*.

Discussion. So I'm trying to prove a relationship between the number of edges and vertices in any possible graph. I can't assume anything about the structure of the graph: it could have any number of vertices and edges, and this property should still hold.

Because the inequality says that $|E|$ is less than or equal to some expression, we can try to figure out what the *maximum* possible number of edges in G is. So the question is: *Given n vertices, how many different edges could there be?*

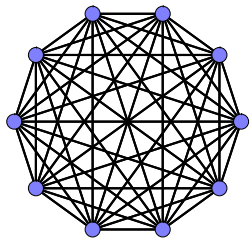
The answer is a straightforward application of the counting work we did earlier: each edge is formed by *choosing* two vertices, where order does not matter, and duplicate edges are not allowed.

Proof. Let $G = (V, E)$ be an arbitrary graph. We want to prove that $|E| \leq \frac{|V|(|V|-1)}{2}$.

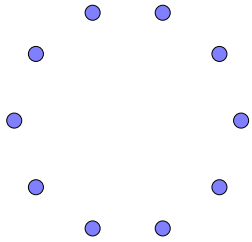
Each edge in G consists of a pair of vertices from V, where order does not matter. There are exactly $\binom{|V|}{2} = \frac{|V|(|V|-1)}{2}$ possible pairs of vertices, and so there are a maximum of this many possible edges.

So $|E| \leq \frac{|V|(|V|-1)}{2}$. □

A graph with all possible edges.



A graph with no edges.



Exercise Break!

5.1 Let $n \in \mathbb{Z}^+$. Find, with proof, the number of distinct graphs with the vertex set $V = \{1, 2, \ldots, n\}$.

We say two such graphs are distinct if one of them has an edge (u, v) and the other one does not have this edge with the same vertices.

Paths and connectedness

Often when we use graphs in modelling the real world, it is not sufficient to capture just a single relationship between entities. Our goal now is to use individual edges, which represent some sort of relationship between vertices, to build up extended, indirect connections between vertices. In a social network, for example, we want to be able to go from friends to "friends of friends," and even "friends of friends of friends of friends." In a graph representing roads between cities, we want to be able to go from "a route between cities using one road" to "a route between cities using n roads." We use the following definitions to make precise these notions of "indirect" relationships.

Definition 5.2 (adjacent, neighbour)**.** Let $G = (V, E)$, and let $v_1, v_2 \in V$. We say that v_1 and v_2 are **adjacent** if and only if there exists an edge between them, i.e., $(v_1, v_2) \in E$. Equivalently, we can also say that v_1 and v_2 are **neighbours**.

Remember that order doesn't matter in the edge pairs.

Definition 5.3 (path, connected, path length)**.** Let $G = (V, E)$ and let $u, u' \in V$. A **path** between u and u' is a sequence of *distinct* vertices $v_0, v_1, v_2, \ldots, v_k \in V$ which satisfy the following properties:

Like edges, paths are directionless; a path from u to u' is also a path from u' to u.

- $v_0 = u$ and $v_k = u'$. (The endpoints of the path are u and u'.)
- Each consecutive pair of vertices are adjacent. (So v_0 and v_1 are adjacent, and so are v_1 and v_2, v_2 and v_3, etc.)

The **length** of a path is one less than the number of vertices in the sequence (so the above sequence would have length k); more intuitively, the length of the path is the number of *edges* which are used by this sequence.

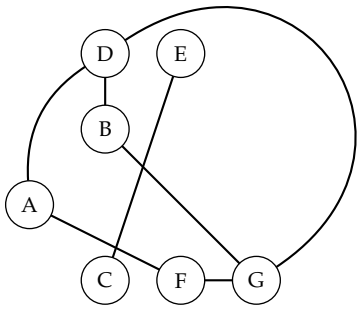
We say that u and u' are **connected** if and only if there exists a path between u and u'.

Note that this is an existentially quantified statement: there could be multiple paths between u and u', and we would still consider u and u' to be connected. Note that there are two different definitions of "connected," one for pairs of vertices and one for graphs. Of course, which definition we mean depends on the type of object we're talking about.

We say that graph G is **connected** if and only if for all pairs of vertices $u, v \in V$, u and v are connected.

Being connected is a fundamental property of graphs. Imagine, for example, a geographical representation where each graph vertex is a city, and each edge a road between two cities. If this graph is not connected, then it is *not* possible to get from one city to any other city by road.

Example 5.3. Consider the graph on the right.



(a) Are the vertices A and B adjacent?

(b) Are the vertices A and B connected?

(c) What is the length of the shortest path between vertices B and F?

(d) Prove that this graph is not connected.

Discussion. (a) through (c) are exercises in understanding the definitions we've just read.

(a) A and B are *not* adjacent: there is no edge between them.

(b) A and B *are* connected: there is a path A, F, G, B between them.

(c) There is a path of length 2 between B and F: B, G, F. How do we know this is the shortest one? The only path of length one which could be between B and F is simply the sequence B, F, which is *not* a path because B and F are not adjacent.

(d) is a bit more complicated, and warrants a formal proof.

Translation. Let us first translate the statement "this graph is not connected." We'll let $G = (V, E)$ refer to this graph (and corresponding vertex and edge sets). So we can write this statement as "G is not connected," but that's not very illuminating. Let us unpack the definition of

connected for graphs, which requires every pair of vertices in the graph to be connected:

This is as much a review of logical manipulation rules as it is of practicing unpacking definitions!

$$
\begin{aligned}
& G \text{ is not connected} \\
\iff\ & \neg(G \text{ is connected}) \\
\iff\ & \neg(\forall u, v \in V,\ u \text{ and } v \text{ are connected}) \\
\iff\ & \exists u, v \in V,\ u \text{ and } v \text{ are } \textit{not} \text{ connected} \\
\iff\ & \exists u, v \in V, \text{ there is no path between } u \text{ and } v
\end{aligned}
$$

We actually went a step further and unpacked the definition of connected for vertex pairs as well. Hopefully this makes it clear what it is we need to show: that there exist two vertices in the graph which do not have a path between them.

Proof. Let $u = B$ and $v = E$ be vertices in the above graph. Then there is no path between u and v, and so this graph is not connected. □

Now let us look at one extremely useful property of connectedness: the fact that if two vertices in a graph are both connected to a third vertex, then they are also connected to each other.

Example 5.4. Let $G = (V, E)$ be a graph, and let $u, v, w \in V$. If v is connected to both u and w, then u and w are connected.

In other words, vertex-connectedness is a *transitive* property.

Translation. Once again, after we get over the fact that we are quantifying over the set of all possible graphs, the translation is pretty straightforward, as the statement's structure is not that complex. It is quite cumbersome to write $\mathcal{G}$, the set of all graphs, every single time. Since how we declare a graph variable looks syntactically different ("$G = (V, E)$") than declaring a numeric variable, we'll adopt an assumed domain of "set of all graphs" for the rest of this chapter. To make the formula even more concise, we'll use the predicate $Conn(G, u, v)$ to mean that "u and v are connected vertices in G."

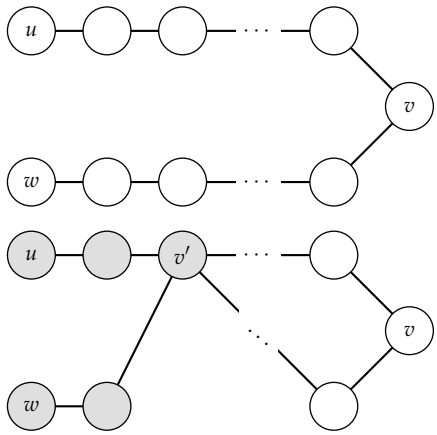
You may do the same in your proofs, but keep in mind that this is only appropriate for graph variables. For variables over other domains, you should still explicitly write every domain.

$$\forall G = (V, E),\ \forall u, v, w \in V,\ (Conn(G, u, v) \land Conn(G, v, w)) \Rightarrow Conn(G, u, w).$$

Discussion. Let's examine the structure of the statement first. We have an arbitrary graph and three vertices in that graph. Because we're proving an implication, we assume its hypothesis: that u and v are connected, and that v and w are connected. We need to prove that u and w are also connected.

Let's rephrase that by unpacking the definition of "connected." We can assume that there is a path between u and v, and between v and w. We need to prove that there is a path between u and w. Phrased that way, it may seem obvious what to do: create a path between u and w by *joining the path between u and v and the one between v and w*.

There's only one problem with this: the paths between u and v and v and w might contain some vertices in common, and paths are not allowed to have duplicate vertices. We can fix this, however, by using a simple idea: find the first point of intersection between the paths, and join them at that vertex instead.



Proof. Let $G = (V, E)$ be a graph, and $u, v, w \in V$. Assume that u and v are connected, and v and w are connected. We want to prove that u and w are connected.

Let P_1 be a path from u to v, and P_2 be a path from v to w. (By the definition of connectedness, both of these paths must exist.)

Handling multiple shared vertices: Let $S \subseteq V$ be the set of all vertices which appear on both P_1 and P_2. Note that this set is not empty, because $v \in S$. Let v' be the vertex in S which is *closest* to u in P_1. This means that *no* vertex in P_1 between u and v' is in S, or in other words, is also on P_2.

Finally, let P_3 be the path which is formed by taking the vertices in P_1 from u to v', and then the vertices in P_2 from v' to w. Then P_3 has no duplicate vertices, and is indeed a path between u and w. By the definition of connectedness, this means that u and w are connected. □

A limit for connectedness

Intuitively, since connectivity is based on paths between vertices, which in turn are built from edges, it is natural to think that we can "force" a graph to be connected by simply adding more edges to it. In this section, we will investigate this by trying to answer the question: "how many edges does it take to ensure that a graph is connected?"

Example 5.5. Let $n \in \mathbb{Z}^+$. There exists a number $M \in \mathbb{Z}^+$ such that for all graphs $G = (V, E)$, if $|V| = n$ and $|E| \geq M$, then G is connected.

Translation. The structure of this statement is a little more complex, but you should be able to handle this with all the work you've previously done. Keep in mind that we have three *alternating* quantifications – n, M, and $G = (V, E)$ – as well as a couple of hypotheses in an implication.

$\forall n \in \mathbb{Z}^+, \exists M \in \mathbb{Z}^+, \forall G = (V, E), (|V| = n \wedge |E| \geq M) \Rightarrow G$ is connected.

Since this is already a little long, we won't unpack the definition of connected here, but be ready to do so in the discussion/proof to follow.
Discussion. There are two important things to note in the statement structure. The first is that because M is existentially-quantified, we get to pick its value. The second is that because this quantification happens *after* n, the value of M is allowed to depend on n. This turns out to be a great power indeed.

For example, if we set $M = n^2$, then because we know that *no* graph exists with n vertices and n^2 or more edges, the implication becomes vacuously true. This is a valid proof, but not that interesting.

see Example 5.2

Instead, let's set $M = \dfrac{n(n-1)}{2}$, i.e., force the graph G to have all possible edges. The proof will still be straight-forward, but at least such a graph exists.

Proof. Let $n \in \mathbb{Z}^+$, let $M = \dfrac{n(n-1)}{2}$, and let $G = (V, E)$ be a graph. Assume that $|V| = n$ and $|E| \geq M$. We need to prove that G is connected.

Because the maximum number of edges in a graph with n vertices is exactly $\dfrac{n(n-1)}{2}$, this means that G must have all possible edges. Then any two vertices $u, v \in V$ are adjacent, and hence connected. So then G is connected. □

Review the definitions of "connected" if you aren't sure about the last two sentences here.

The previous example shows the danger of making statements using existential quantifiers: often it is easy to prove that a particular value exists, but what we really care about is the "best" possible value. We don't want just any M, but the smallest possible one which forces a graph to be connected. For instance, it would be much more interesting if we could prove the following statement:

$$\forall n \in \mathbb{Z}^+,\ \forall G = (V, E),\ (|V| = n \wedge |E| \geq 2n) \Rightarrow G \text{ is connected.}$$

Unfortunately, this statement is false, and in fact the value $M = 2n$ is not even close, as we'll prove next.

Example 5.6. Let $n \in \mathbb{Z}^+$. Then there exists a graph $G = (V, E)$, such that $|V| = n$ and $|E| = \dfrac{(n-1)(n-2)}{2}$, and G is not connected.

Translation.

$$\forall n \in \mathbb{Z}^+,\ \exists G = (V, E),\ |V| = n \wedge |E| = \frac{(n-1)(n-2)}{2} \wedge G \text{ is not connected.}$$

Discussion. This statement looks a little different than the one from the previous exampls, but in fact is essentially its negation. Here, we

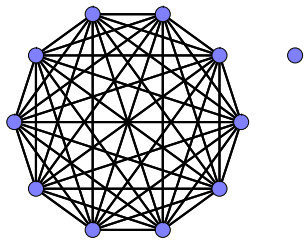
More precisely, the parts after the $\forall n \in \mathbb{Z}^+$ are negations of each other.

are asked to show that for any n, there is a graph with n vertices and $\frac{(n-1)(n-2)}{2}$ edges, but which is still not connected.

Okay, so how do we prove this? Note that this time we can choose the graph, though we are constrained by the number of vertices and edges the graph must have. The form $\frac{(n-1)(n-2)}{2}$ is a big hint, since we know that this is equal to $\binom{n-1}{2}$, which looks suspiciously like the maximum number of edges on $n-1$ vertices...

Proof. Let $n \in \mathbb{Z}^+$. Let $G = (V, E)$ be the graph defined as follows:

This is the first time we're defining a graph in a proof, rather than assuming the graph is arbitrary!



- $V = \{v_1, v_2, \ldots, v_n\}$.
- $E = \{(v_i, v_j) \mid i, j \in \{1, \ldots, n-1\}\}$. That is, E consists of all edges between the first $n-1$ vertices, and has no edges connected to v_n.

We need to now show three things:

(i) $|V| = n$.

(ii) $|E| = \frac{(n-1)(n-2)}{2}$.

(iii) G is not connected.

For (i), we have explicitly labelled the n vertices in V, and so it is clear that $|V| = n$.

For (ii), we have chosen all possible pairs of vertices from $\{v_1, v_2, \ldots, v_{n-1}\}$ for the edges. There are exactly $\binom{n-1}{2} = \frac{(n-1)(n-2)}{2}$ such edges.

For (iii), because v_n is not adjacent to any other vertex, it cannot be connected to any other vertex. So G is not connected. □

So we have now proved that a graph with a fairly large number of edges can still not be connected. It is worth noting that $\frac{(n-1)(n-2)}{2} = \frac{n(n-1)}{2} - (n-1)$. That is, there is a graph which is missing only $n-1$ edges from the set of all possible of edges, but is still not connected. The question becomes: can we go higher still? Is is possible for a graph on n vertices to have more than $\frac{(n-1)(n-2)}{2}$ edges and yet still be not connected? Or is the best possible M from our original question indeed $\frac{(n-1)(n-2)}{2} + 1$?

It turns out that the latter is true, and this will be the last – and most challenging – proof we do in this section.

Example 5.7. Let $n \in \mathbb{Z}^+$. For all graphs $G = (V, E)$, if $|V| = n$ and $|E| \geq \frac{(n-1)(n-2)}{2} + 1$, then G is connected.

Translation.

$$\forall n \in \mathbb{Z}^+, \forall G = (V, E), \left(|V| = n \wedge |E| \geq \frac{(n-1)(n-2)}{2} + 1\right) \Rightarrow G \text{ is connected.}$$

Discussion. So we are back to our orginal example, except now the M has been picked for us, and we are using an edge number of $\frac{(n-1)(n-2)}{2} + 1$. It is tempting for us to base our proof on the previous example: after all, if we start with a graph that has $n-1$ of its vertices all adjacent to each other, and then add one more edge to the remaining vertex, the new graph is certainly connected. However, this line of thinking relies on a particular starting point for the structure of G, which we cannot assume anything about (other than the number of vertices and edges, of course).

The problem is that even with these restrictions on the number of edges and vertices, it is hard to conceptualize enough common structure among such graphs to use in a proof.

If that's too abstract, just imagine trying to complete the following statement: "Every graph with n vertices and at least $\frac{(n-1)(n-2)}{2} + 1$ edges is/has _____."

What is more promising, though, is trying to take a graph which satisfies the the constraints on its number of edges and vertices, and then remove a vertex to make the graph smaller, and argue two things:

- the smaller graph is connected
- the vertex we removed is adjacent to at least one vertex in the smaller graph

This idea of "removing a vertex" from a graph to make the problem smaller and simpler can be formalized using induction, and is in fact one of the most common proof strategies when dealing with graphs. The one thing to keep in mind here is that we're doing induction on n, but the predicate we need to prove – contains quantifiers, making it more complex.

We weren't kidding about the great usefulness of induction.

You'll notice that the inductive step in this proof is more complicated, and is split up into cases, and involves a sub-proof inside. As you read through this proof, look for both the *structure* as well as *content* of the proof: both are vital to understand.

Proof. We will proceed by induction on n. More precisely, define the following predicate over the positive integers:

$$P(n) : \forall G = (V, E), \left(|V| = n \wedge |E| \geq \frac{(n-1)(n-2)}{2} + 1\right) \Rightarrow G \text{ is connected.}$$

In words, $P(n)$ says that for every graph G with n vertices and at least $\frac{(n-1)(n-2)}{2}+1$ edges, G must be connected. We want to prove that $\forall n \in \mathbb{Z}^+,\ P(n)$ using induction.

Base Case: $n = 1$. This is a good exercise in substitution:

$$P(1) : \forall G = (V, E),\ (|V| = 1 \wedge |E| \geq 1) \Rightarrow G \text{ is connected}$$

$\frac{(1-1)(1-2)}{1}+1 = 1$

This statement is vacuously true: no graph exists that has only one vertex and at least one edge, since an edge requires two vertices.

Inductive Step: Let $k \in \mathbb{Z}^+$, and assume that $P(k)$ holds. We need to prove that $P(k+1)$ also holds, i.e.:

$$P(k+1) : \forall G = (V, E),\ \left(|V| = k+1 \wedge |E| \geq \frac{k(k-1)}{2}+1\right) \Rightarrow G \text{ is connected.}$$

Let $G = (V, E)$, and assume that $|V| = k+1$ and $|E| \geq \frac{k(k-1)}{2}+1$. We now need to prove that G is connected. We will split up this proof into two cases.

Case 1: Assume $|E| = \frac{(k+1)k}{2}$, i.e., G has all possible edges. In this case, G is certainly connected.

Case 2: Assume $|E| < \frac{(k+1)k}{2}$. We now need to prove the following claim:

Claim. There exists a vertex in G which has between 1 and $k-1$ neighbours, inclusive.

Remember that there are $k+1$ vertices in total, so the maximum number of neighbours is k. This claim is saying that there exists a vertex which has at least one neighbour, but not the maximum number.

Proof of claim. Since G has fewer than the maximum number of possible edges, there exists a vertex pair (u, v) which is *not* an edge. Both u and v have at most $k-1$ neighbours, since there are $k-1$ vertices in G other than these two.

We leave showing that both u and v have at least one neighbour as an exercise. □

Using this claim, we let v be a vertex which has at most $k-1$ neighbours. Let $G' = (V', E')$ be the graph which is formed by taking G and removing v from V, and all edges in E which use v. Then $|V'| = |V| - 1 = k$, i.e., we've decreased the number of vertices by 1. This is good because we're trying to do induction on the number of vertices.

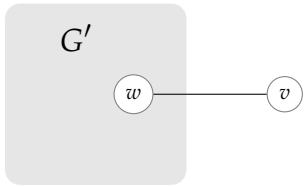
However, in order to use $P(k)$, we need not just that the number of

Remember that $P(k)$ states an implication: *if* the graph has the appropriate number of vertices and edges, *then* it is connected.

vertices to be k, but that the number of edges is at least $\frac{(k-1)(k-2)}{2} + 1$. This is what we'll show next.

$$\begin{aligned} |E'| &= |E| - \#\text{ of removed edges} \\ &\geq |E| - (k-1) && \text{(at most } k-1 \text{ edges removed)} \\ &\geq \frac{k(k-1)}{2} + 1 - (k-1) && \text{(assumption on } |E|) \\ &= \frac{(k-2)(k-1)}{2} + 1 \end{aligned}$$

Now that we have this, we can finally use the induction hypothesis: since $|V'| = k$ and $|E'| \geq \frac{(k-2)(k-1)}{2} + 1$, we conclude that G' is connected.



Finally, let us use the fact that G' is connected to show that G is also connected. First, any two vertices not equal to v are connected in G because they are connected in G'. What about v, the vertex we removed from G to get G'? Recall our claim: v has at least one neighbour, so call it w. Then v is connected to w, but because G' is connected, w is connected to every other vertex in G. By Example 5.4 (the transitivity of connectedness) , we know that v must be connected to all of these other vertices, and so G is connected. □

Exercise Break!

5.2 These questions concern the proof that we just saw.

(a) Let $n \in \mathbb{Z}^+$, and let $G = (V, E)$ be a graph. Prove that if $|V| = n$ and $|E| \geq \frac{(n-1)(n-2)}{2} + 1$, then every vertex in G has at least one neighbour.

(b) It may have struck you as a little strange that we used cases in our proof of the inductive step.
What goes wrong with the argument in the second case if we try to include the case when G has all $\frac{(k+1)k}{2}$ possible edges? (Hint: this is actually quite subtle, and took us a while to pinpoint ourselves!)

Cycles and trees

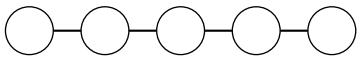
We spent the last section investigating how many edges a graph would need to force it to be connected. We will now turn to the dual question:

Or, how many edges are *sufficient* for connectedness.

how many edges is a graph forced to have if it is connected? Rather than taking a graph and adding edges to it to see how far we can go without it becoming connected, we now ask how many edges can we *remove* from a connected graph without disconnecting it.

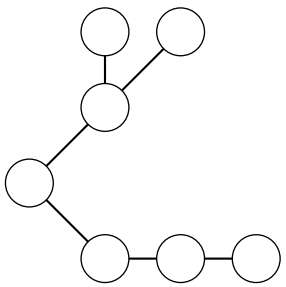
Or, how many edges are *necessary* for connectedness.

We might consider some simple examples to gain some intuition here. For example, suppose we have a graph with n vertices which is just a path.



This has $n-1$ edges, and if you remove any edge from it, the resulting graph will be disconnected (we leave a proof of this as an exercise).

But this isn't the only possible configuration for such a graph. The one on the right certainly isn't a path; you may recognize it as a "tree," though we won't define this term formally until later in this chapter.



Indeed, removing any edge from this graph disconnects it, and you might notice by counting that the number of edges is again one less than the number of vertices.

It turns out that these examples do give us the right intuition: *any* connected graph $G = (V, E)$ must have $|E| \geq |V| - 1$. The tricky part is proving this. Once again, we must struggle with the fact that even though the previous examples gave us some intuition, it is a challenge to generalize these examples to obtain an argument that works on all graphs satsifying these vertex and edge counts.

The contrapositive is also an interesting statement: if a graph has at least two fewer edges than vertices, it cannot be connected.

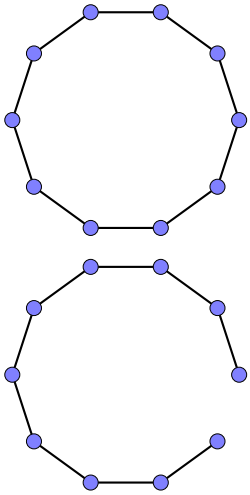
To get a formal proof, we'll need some way of characterizing exactly when we can remove an edge from a graph without disconnecting it. The following definition is an excellent start.

Definition 5.4 (cycle)**.** Let $G = (V, E)$ be a graph. A **cycle** in G is a sequence of vertices $v_0, \ldots, v_k$ satisfying the following conditions:

- $k \geq 3$
- $v_0 = v_k$, and all the other vertices are distinct from each other and v_0
- each consecutive pair of vertices is adjacent

In other words, a cycle is like a path, except it starts and ends at the same vertex. The length of a cycle is the number of edges used by the sequence, which is also the number of distinct vertices in the sequence (the above notation describes a cycle of length k). Cycles must have length at least 3; two adjacent vertices are not considered to form a cycle.

To use our example of cities and roads, if there is a cycle in the graph, it is possible to make a trip which starts and ends at the same city, and travels no road or city more than once during the trip.



Getting back to our motivation, cycles are a form of "connectedness redundancy" in a graph. Vertices in a cycle are all obviously connected to each other, but even if one edge is removed, the result is a path. In this case, the cycle's vertices are still connected to each other – albeit with a possibly much longer path to travel. Even though the diagrams on the right illustrate this property for a cycle itself, we will now show that this property holds even when this cycle is part of a larger graph.

Example 5.8. Let $G = (V, E)$ be a graph. Assume that G is connected and contains a cycle. Let e be any edge which is in a cycle of G. Then the graph obtained by removing e from G is still connected.

Translation. There are a lot of quantified variables here, and some assumptions which are perhaps not obvious from the English. It is certainly a worthwhile exercise to translate this statement explicitly. The trickiest part is the condition on e (that it is part of a cycle of G); remember that we generally represent such conditions as assumptions in a logical implication.

Again for brevity, we will use the notation $G - e$ to represent the graph obtained by removing edge e from G.

$$\forall G = (V, E),\ \forall e \in E,\ \ (G \text{ is connected} \wedge G \text{ contains a cycle} \wedge e \text{ is in a cycle of } G) \Rightarrow G - e \text{ is connected.}$$

Discussion. This is a statement about a particular transformation: if we start with a connected graph and remove an edge in a cycle, then the resulting graph is still connected.

We get to assume that the original graph is connected and has a cycle, but that's it. We don't know anything else about the graph's structure, nor even which edge we picked in relation to the graph.

That said, it seems like we should be able to simply make an argument based on the transitivity of connectedness (Example 5.4: if we remove the edge (u, v) from the cycle, then we already know that u and v are still connected, so all the other vertices should still be connected too).

Proof. Let $G = (V, E)$ be a graph, and $e \in E$ be an edge in the graph. Assume that G is connected, that G contains a cycle, and that e is in a cycle. Let $G' = (V, E \backslash \{e\})$ be the graph formed from G by removing edge e. We want to prove that G' is also connected, i.e., that any two vertices in V are connected in G'.

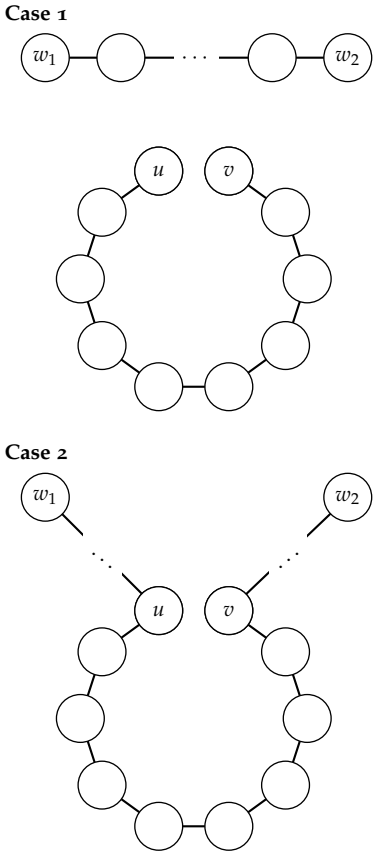
Let $w_1, w_2 \in V$ be two vertices. By our assumption, we know that w_1 and w_2 are connected in G. We want to show that they are also connected in G', i.e., there is a path in G' between w_1 and w_2.

Let P be a path between w_1 and w_2 in G (such a path exists by the definition of connectedness). We divide our proof into two cases: one where P uses the edge e, and another where it does not.

Case 1: P does not contain the edge e. Then P is a path in G' as well (since the only edge that was removed is e).

Case 2: P does contain the edge e. Let u be the endpoint of e which is closer to w_1 on the path P, and let v be the other endpoint.

This means that we can divide the path P into three parts: P_1, the part from w_1 to u, the edge (u, v), and then P_2, the part from v to w_2. Since P_1 and P_2 cannot use the edge (u, v) – no duplicates – they must be paths in G' as well. So then w_1 is connected to u in G', and w_2 is connected to v in G'. But we know that u and v are also connected in G' (since they were part of the cycle), and so by Example 5.4, w_1 and w_2 are connected in G'. □



This example tells us that if we have a connected graph with a cycle, it is always possible to remove an edge from the cycle and still keep the graph connected. Since we are interested in talking about the minimum number of edges necessary for connecting a graph, we'll now think about graphs which don't have any cycles.

Definition 5.5 (tree)**.** A **tree** is a graph which is connected and has no cycles.

We would like to say that trees are the "minimally-connected" graphs: that is, the graphs which have the fewest number of edges possible but are still connected. It may be tempting to simply assert this based on the definition and what we have already proven, but let G be a connected graph, and consider the following statements carefully:

(i) *If* G has a cycle, *then* there exists an edge in G such that when it is removed, the resulting graph is still connected.

(ii) *If* G is a tree, *then* it does not have a cycle.

(iii) *If* G does not have a cycle, *then* there does not exist an edge in G such that when it is removed, the resulting graph is still connected.

We know that (i) is true, by Example 5.8. (ii) is true simply by the definition of "tree." How do we know (iii) is true?

In fact, we don't. The statements (i) and (iii) may look very similar, but they are *not* logically equivalent. In fact, (iii) is logically equivalent to the *converse* of (i): if we let P be the statement "G has a cycle" and Q be the statement "there exists an edge in G such that when it is removed,

the resulting graph is still connected," then (i) is simply $P \Rightarrow Q$, while (iii) is $\neg P \Rightarrow \neg Q$.

So we actually need to prove (iii) directly, which is what we'll do next.

Example 5.9. Let G be a connected graph. If G does not have a cycle, then there does not exist an edge in G such that when it is removed, the resulting graph is still connected.

Translation. Once again, we use the notation $G - e$ to represent the graph obtained from G by removing edge e.

$$\forall G = (V, E),\ \big(G \text{ is connected} \wedge G \text{ does not have a cycle}\big) \Rightarrow \neg(\exists e \in E,\ G - e \text{ is connected}).$$

In general, having to prove that there does *not* exist some object satisfying some given conditions is challenging; it is often easier to assume such an object exists, and then prove that its existence violates one or more of the given assumptions. This can be formalized by writing the *contrapositive* of our original statement.

$$\forall G = (V, E),\ (\exists e \in E,\ G - e \text{ is connected}) \Rightarrow \big(G \text{ is not connected} \vee G \text{ has a cycle}\big).$$

Discussion. So we can *assume* that there exists an edge e with this nice property that removing it keeps G connected. From this, we need to prove that G is not connected, or G has a cycle.

Unfortunately, if $G - e$ is connected, then G must also be connected (since adding edges cannot disconnect any vertices). So we need to prove that G has a cycle. Note that we only need to show that a cycle exists – it may or may not have anything to do with e, but it is probably a good bet that it does.

If we need to prove $P \vee Q$, but we know that P is false, then our only hope is to prove Q.

The key insight is that if we remove e, we remove one possible path between its endpoints. But since the graph must still be connected after removing e, there must be another path between its endpoints.

Proof. Let $G = (V, E)$ be a graph. Assume that there exists an edge $e \in E$ such that if e is removed from G, the graph is still connected. We want to prove that G has a cycle.

Let $G' = (V, E \backslash \{e\})$ be the graph obtained by removing e from G. Then G' must be connected, by our assumption.

Let u and v be the endpoints of e. There is a path P in G' from u to v which does not use e (since e isn't in G'). Then G has a cycle: take the path P, then add the edge e to it. □

Thus we now can state and prove the following fact about trees.

Example 5.10. Let G be a tree. Then removing any edge from G disconnects the graph.

Proof. This follows directly from the previous claim. By definition, G does not have any cycles, and so there does not exist an edge that can be removed from G without disconnecting it. □

We can say that a tree is the "backbone" of a connected graph. While a connected graph may have many edges and many cycles, it is possible to identify an underlying tree structure in the graph which, if it remains unchanged, ensures that the graph remains connected, regardless of what other edges are removed.

This insight is the basis of computing *minimum spanning trees*, a well-studied problem in computer science that you will about learn in future courses.

Now, let us return to our original motivation of counting edges to prove the following remarkable result, which says that the number of edges in a tree depends *only* on the number of vertices.

Theorem 5.1. *Let $G = (V, E)$ be a tree. Then $|E| = |V| - 1$.*

Translation.

$$\forall\ G = (V, E),\ G \text{ is a tree} \Rightarrow |E| = |V| - 1.$$

Discussion. We have previously observed that this property seems to hold on trees that we drew ourselves. But of course this is not a formal proof, since we cannot assume anything about the particular structure of a tree.

A natural alternate strategy is to take a tree, remove a vertex from it, and use induction to show that the resulting tree satisfies this relationship between its numbers of vertices and edges.

This only works, though, if we can pick a vertex whose removal from G results in a tree – and in particular, results in a connected graph. To do this, we need to pick a vertex that is at the "end" of the tree.

Rather than proceeding with the proof directly, we recognize that a likely claim we'll need to use in our proof is that picking such an "end" vertex is always possible. Rather than embedding a subproof within the main proof, we will do it separately first.

Lemma 5.2. *Let $G = (V, E)$ be a tree. If $|V| \geq 2$, then G has a vertex which has exactly one neighbour.*

Translation.

$\forall G = (V, E),\ (G \text{ is a tree} \wedge |V| \geq 2) \Rightarrow (\exists v \in V,\ v \text{ has exactly one neighbour})$.

Discussion. What does it mean for a vertex to have exactly one neighbour? Intuitively, it means that we're at the "end" of the tree, and can't go any further. This makes sense on a visual diagram, but how can we formalize this? Suppose we start at an arbitrary vertex, and traverse edges to try to get as far away from it as possible. Because there are no cycles, we cannot revisit a vertex. But the path has to end somewhere, so it seems like that its endpoint must have just one neighbour.

Proof. Let $G = (V, E)$ be a tree. Assume that $|V| \geq 2$. We want to prove that there exists a vertex $v \in V$ which has exactly one neighbour.

Let u be an arbitrary vertex in V. Let v be a vertex in G which is at the maximum possible distance from u, i.e., the path between v and u has maximum possible length (compared to paths between u and any other vertex). Let P be the shortest path between v between u. We will prove that v has exactly one neighbour.

We know that v has *at least* one neighbour: the vertex immediately before it on the path to u. v cannot be adjacent to any other vertex on P, as otherwise G would have a cycle. Also, v cannot be adjacent to any other vertex w *not* on P, as otherwise we could extend P to include w, and this would create a longer path.

And so v has exactly one neighbour (the one on P immediately before v). □

With this lemma in hand, we can now give a complete proof of Theorem 5.1. The key will be to use induction, removing from the original graph a vertex with just one neighbour, so that the number of edges also only changes by one. But how can we use induction on a statement that starts with $\forall\ G = (V, E)$? We are used to seeing induction used with a statement of the form $\forall\ n \in \mathbb{N}$ or $\forall\ n \in \mathbb{Z}^+$. To this end, we introduce a variable n to stand for the number of vertices in a graph, and then apply induction using the number of vertices. The statement that we will prove becomes

$$\forall\ n \in \mathbb{Z}^+,\ \forall\ G = (V, E),\ (G \text{ is a tree} \wedge |V| = n) \Rightarrow |E| = n - 1.$$

Proof of Theorem 5.1. We will proceed by induction on n, the number of vertices in the tree. Let $P(n)$ be the following statement (over positive integers):

$$P(n) : \forall G = (V, E),\ (G \text{ is a tree} \wedge |V| = n) \Rightarrow |E| = n - 1.$$

We want to prove that $\forall n \in \mathbb{Z}^+,\ P(n)$.

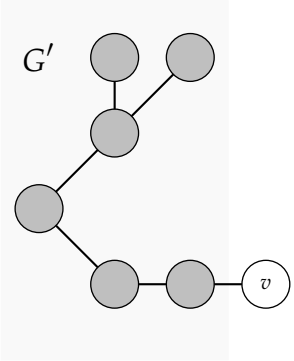
Base Case: $n = 1$. In this case, G has only a single vertex, and cannot have any edges. Then $|E| = 0 = n - 1$.

Inductive Step: Let $k \in \mathbb{Z}^+$, and assume that $P(k)$ is true, i.e., for all graphs $G = (V, E)$, if G is a tree and $|V| = k$, then $|E| = k - 1$. We want to prove that $P(k+1)$ is also true. Unpacking $P(k+1)$, we get:

$$\forall G = (V, E),\ (G \text{ is a tree} \wedge |V| = k+1) \Rightarrow |E| = k.$$

So let $G = (V, E)$ be a tree, and assume $|V| = k + 1$. We want to prove that $|E| = k$.

By Lemma 5.2, there exists a vertex $v \in V$ which has exactly one neighbour. Let $G' = (V', E')$ be the graph obtained by removing v and the one edge on v from G. Then $|V'| = |V| - 1 = k$ and $|E'| = |E| - 1$.



You will prove this in one of your exercises.

We know that G' is also a tree. Then the induction hypothesis applies, and we can conclude that $|E'| = |V'| - 1 = k - 1$.

This means that $|E| = |E'| + 1 = k$, as required. □

Combining everything together, we can conclude the following required number of edges for any connected graph.

Since every connected graph contains a tree (just keep removing edges in cycles until you cannot remove any more), this constraint on the numbers of edges in a tree translates immediately into a lower bound on the number of edges in any connected graph (in terms of the number of vertices of that graph).

Theorem 5.3. *Let $G = (V, E)$ be a graph. If G is connected, then $|E| \geq |V| - 1$.*

Exercise Break!

5.3 Adapt the proof of Lemma 5.2 to prove that for any tree $G = (V, E)$, if $|V| \geq 2$ then G has at least *two* vertices with exactly one neighbour.

5.4 Prove the following claim. Let $G = (V, E)$ be a tree, and let v be a vertex in G which has exactly one neighbour. Prove that the graph obtained by removing v from G is also a tree.

Rooted trees

The definition of "tree" that we have used so far – a connected graph with no cycles – is actually more general than what you may be familiar with from typical computer science applications. This is because trees themselves do not enforce an orientation or ordering amongst vertices,

while in practice almost all of their uses involve a notion of hierarchy that elevates some vertices above others.

For this type of application, we specialize our more general definition to add this notion of hierarchy. Note that this definition is a "cosmetic" one in the sense that it does not actually say anything different about the *structure* of a graph, but merely how we interepret the vertices of the graph.

Definition 5.6 (rooted tree, root)**.** A **rooted tree** is a tree which has one vertex labelled as the **root** of the tree.

So when you hear the average computer scientist talking about trees, they're really talking about rooted trees.

Simply by designating one vertex in a tree as special, we immediately obtain a sense of direction in the tree; we can now use distance from the root as a partial ordering of the vertices, and talk about moving "away from the root" or "towards the root" when traversing edges. We typically represent this sense of direction visually by drawing rooted trees with the root vertex at the top, although of course you should keep in mind that this is merely a convention.

We will now introduce some new terminology that emerge naturally from this orientation. Note that much of the terminology matches our intuition for relationships among relatives in a family tree.

Definition 5.7 (parent, child, ancestor, descendent, leaf)**.** Let $G = (V, E)$ be a rooted tree, and $r \in V$ be the root of the tree. Let $v \in V$ be an arbitrary vertex (including, but not limited to, r itself).

The **parent** of v is its neighbour which is closer to r than v is. A **child** of v is any of its other neighbours (which are further from r than v is).

Equivalently, the parent is the vertex immediately before v on the path from r to v.

An **ancestor** of v is any vertex on the path between r and v, not including v itself. (Equivalently, an ancestor of v is its parent, its parent's parent, its parent's parent's parent, etc.)

A **descendant** of v is any vertex w such that v is on the path between r and w. (Equivalently, a descendant of v is its child, its child's child, its child's child's child, etc.)

A **leaf** of a rooted tree is any vertex which has no children.

Note that all leaves of a rooted tree have at most one neighbour. Lemma 5.2 can be used to show that each rooted tree has at least one leaf.

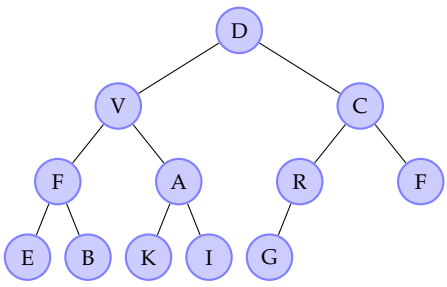
Example 5.11. Consider the rooted tree on the right.

(a) What is the parent of A?

(b) What are the children of C?

(c) What are the ancestors of B?

(d) What are the ancestors of D?



(e) What are the descendents of C?

(f) What are the descendents of B?

Discussion. This is another simple check on the terminology.

The only ones of note are (d) and (f). Since vertex D is the root of the tree (remember the convention of drawing the root of the tree at the top of the diagram), it has *no* ancestors, and similarly, because B is a leaf, it has no descendents.

Definition 5.8 (height)**.** The **height** of a rooted tree is one plus the length of the longest path between the root and a leaf. The "one plus" is to ensure that we are counting vertices instead of edges – e.g., a tree which consists of just the root vertex has height 1, not height 0.

Many books define height as just the length of the longest path, which counts edges rather than vertices. It doesn't make a big difference, but counting vertices makes some of our future calculations look a little cleaner.

We have already studied the relationship between the numbers of vertices and edges in connected graphs. This question is far less interesting when it comes to trees, because there is an exact relationship between the number of vertices and edges in a tree ($|E| = |V| - 1$).

But for rooted trees, we get another fundamental relationship to study: how the the number of vertices influences the height of the tree. This is a question which is fundamental to many computer science applications of rooted trees, which typically traverse a tree by starting at its root and going down. Such algorithms take a longer amount of time depending on how tall the tree is.

Theorem 5.4. *Let $n \in \mathbb{N}$, and assume $n \geq 2$. Then the following statements hold.*

(i) Every rooted tree with n vertices has height ≥ 2.

(ii) There exists a rooted tree with n vertices with height equal to 2.

(iii) Every rooted tree with n vertices has height $\leq n$.

(iv) There exists a rooted tree G with n vertices with height equal to n.

Discussion. Note that there are four different things to prove here. Two of them are universally-quantified statements, establishing universal bounds on the height of *any* rooted tree. Two of them are existentially-quantified statements, saying that the proposed bounds are tight, i.e., they can be met exactly.

These proofs are not very challenging, and we'll leave them as an exercise.

Hint: think about the "extremes" of possible tree structures.

What is more interesting, and what is often done in practice, is to try to restrict the structure of a rooted tree by restricting the number

of children each vertex can have. The following definition is one of the most common such restrictions.

Definition 5.9 (binary rooted tree)**.** A **binary rooted tree** is a rooted tree where every vertex has at most two children.

This means each vertex has at most *three* neighbours in total: one parent, two children.

Our last proof in this course is captures one such relationship between height and number of vertices in binary rooted trees.

Example 5.12. Let $h \in \mathbb{N}$. Let $G = (V, E)$ be a binary rooted tree, and assume that the height of G is $\leq h$. Then $|V| \leq 2^h - 1$.

Translation.

$\forall h \in \mathbb{N}$, $\forall G = (V, E)$, (G is a binary rooted tree $\wedge$ G has height $\leq h$) $\Rightarrow |V| \leq 2^h - 1$.

Discussion. The key insight here is that binary rooted trees are themselves composed of smaller binary rooted trees. If we take G and remove its root, then we get obtain two binary rooted trees, both of which have height $\leq h - 1$.

We should then be able to use induction to prove the inequality.

Proof. We will prove this statement by induction on h. More precisely, let $P(h)$ be the statement that for every binary rooted tree $G = (V, E)$ of height $\leq h$, $|V| \leq 2^h - 1$.

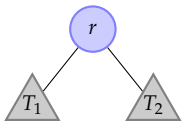
Base Case: $h = 0$. In this case, the only binary rooted tree of height 0 is empty, i.e., has no vertices. Then $|V| = 0$ and $2^h - 1 = 0$, so the inequality holds.

Inductive Step: Let $k \in \mathbb{N}$, and assume that $P(k)$ holds. We want to prove that $P(k+1)$ is also true. More precisely, we can write:

$P(k+1) : \forall G = (V, E)$, ($G$ is a binary rooted tree $\wedge$ G has height $\leq k+1$) $\Rightarrow |V| \leq 2^{k+1} - 1$.

So let $G = (V, E)$ be a binary rooted tree which has height $\leq k+1$. We will show that $|V| \leq 2^{k+1} - 1$.

Let $r \in V$ be the root of G. Consider what happens when we remove r from G. We are left with two smaller binary rooted trees, $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$. Note that one or both of these trees could be empty (i.e., have no vertices or edges), and this is perfectly acceptable.



Since these two trees have height at most k, the induction hypothesis applies: $|V_1| \leq 2^k - 1$ and $|V_2| \leq 2^k - 1$.

Then $|V| = |V_1| + |V_2| + 1$ (the number of vertices in each of the two smaller trees, plus the root):

$$
\begin{aligned}
|V| &= |V_1| + |V_2| + 1 \\
&\leq (2^k - 1) + (2^k - 1) + 1 \\
&= 2 \cdot 2^k - 1 \\
&= 2^{k+1} - 1
\end{aligned}
$$

□

6 *Looking Ahead*

There are many beautiful ideas in Computer Science that make fundamental use of mathematical expression and reasoning. While we cannot do justice to these topics in these notes (many of them are deep), we would like to give you a glimpse of the power of mathematical reasoning in Computer Science. You will learn these and other topics in depth in other Computer Science courses at University of Toronto, including CSC236, CSC263, CSC373, CSC438, CSC448, and CS473.

Turing's legacy: the limitations of computation

What are the limits of computation? Are there functions that we want to get a computer to calculate but that are beyond the capability of computers? This abstract and fuzzy question was formalized precisely by Alan Turing even before computers were invented! Namely, he defined a **Turing machine**, which is a purely mathematical model of computation. It is simple enough to reason about, yet powerful enough to capture any conceivable computational device!

After defining Turing machines, Turing proved that there are important problems that cannot be computed by any Turing machine. Because of the universality of the Turing machine, this then implies that these problems cannot be solved on *any* computer!

Before we try to explain the main ideas behind the proof, we would like to point out that mathematical expression is fundamental to even formulate the question. The abstraction of computation via the mathematical Turing machine model is essential to express a statement that talks about whether a given function can be computed.

The most famous problem that cannot be solved by any Turing machine (and thus by any computer) is called the *Halting Problem*. Informally, the input to the Halting Problem is a program, P, written in some programming language, together with an input to the program, x. The Halting Problem should accept the pair (P, x) if and only if program P

halts on input x. The obvious way to try to solve the Halting Problem on input (P, x) is to simply run or simulate P on the input x and see what happens. If P does halt on x, then our simulation will also halt and we will eventually discover that P halts on x. But what happens when P does not halt on x? In this case we are in trouble! What Turing proved is that it is basically impossible for a computer program to figure out with certainty whether an arbitrary program P will halt on a particular input x. That is, there is no clairvoyant way to examine a program to determine whether not it will halt on an input. Essentially the only thing that one can do is to run the program and see what happens.

By *halt* we mean that if we had an unbounded amount of memory, then running P on x would eventually halt – that is, if it would not get into any infinite loops.

This is a *worst-case* result: there is no procedure that can decide for *all* programs P and for *all* inputs x whether or not P halts on x. But in special cases, it may be easy to determine what will happen.

We will focus on *decision problems*; that is, on problems that compute functions f from the natural numbers to $\{0, 1\}$. Since we want to prove a negative result, we can pick any problem that we'd like, so we aren't cheating by focusing on decision problems. Furthermore, we will assume that the input to our decision problem is encoded in binary, so the input is just some finite-length string of zeroes and ones, and the output is either 0 (false) or 1 (true). We are going to try to explain the main ideas behind the halting problem without getting into too much notation.

Indeed, decision problems turn out to be powerful enough anyway! That is, if f is any function from the natural numbers to the natural numbers, then there is a corresponding decision problem such that this decision problem can be computed if and only if the original function can be computed.

First, we have to define our formal model of computation, the Turing machine (TM). We won't go into any details of Turing machines. They are a beautiful abstraction of computation, but these details aren't really necessary to understand the main thing that we want to prove in this chapter – that certain natural and important functions are beyond the power of computation. The only thing that you will need to know about Turing machines is that they are just programs in a simple programming language where we will assume an unbounded amount of computational memory. If M is a TM for computing a decision problem, it takes as input an arbitrary natural number, encoded by a binary string, s. For each s, the TM may or may not halt on s. If it does halt, then it outputs either 0 (reject) or 1 (accept). Turing machines satisfy the following important properties:

1. Turing machines are a *universal* model of computation – any program written in any standard programming language can be converted to an equivalent Turing machine (TM) program.
2. Turing machines can be enumerated.

By enumerated we mean that there is an algorithm that on input i can output the first i TM's, $M_1, M_2, \ldots, M_i$.

Both of these properties are not unreasonable – if you think of your favorite programming language, such as Python, it should be clear that both of these properties hold.

The first main idea is to come up with *one* explicit decision problem that cannot be computed by a TM. This first problem will not be the

Halting Problem but will instead be a problem that we will construct to make the proof easier for us. By property (2) above, TM programs can be enumerated, so let us write them as $M_1, M_2, \ldots$, where M_i is the i^{th} TM in the enumeration. Now consider the following decision problem, called $\mathcal{D}$ (for the diagonal language): The input to $\mathcal{D}$ is, as usual, a natural number i (encoded in binary). The output is 1 if *either* M_i does not halt on input i, or if M_i halts and outputs 0 on input i. Otherwise, if M_i halts and outputs 1 on i, then D on i outputs 0. In other words, $\mathcal{D}$ does the *opposite* of what M_i does on input i – if M_i rejects i (either by not halting or by halting and not accepting), then $\mathcal{D}$ accepts i, and if M_i accepts input i, then $\mathcal{D}$ rejects i. The very cool thing is that we can prove that the decision problem $\mathcal{D}$ is not computed by any TM!! Why is this? We want to prove that for every $j \in \mathbb{N}$, that M_j does not compute $\mathcal{D}$. So fix some arbitrary $j \in \mathbb{N}$, and consider M_j on input j – by construction it does the opposite thing that $\mathcal{D}$ does on input j, and therefore M_j does not compute $\mathcal{D}$. Since we have proven this for every j, it follows that there is *no* Turing machine that computes $\mathcal{D}$!

The proof method is called *diagonalization* and was first used by Cantor in order to argue there is no bijective mapping from the natural numbers to the real numbers.

Ok so thus far we have found one explicit decision problem, $\mathcal{D}$, that cannot be computed by any TM. Now we want to prove that some specific decision problem (the Halting Problem) also cannot be computed by any TM. At this point, we need to be more precise about what we mean by the Halting Problem. We define the Halting Problem $\mathcal{H}$, as follows. The input is a pair (i, j) where both i and j are natural numbers. The output should be 1 (accept) if M_i halts on input j, and should be 0 (reject) otherwise.

The main point here is that the set of all functions from the natural numbers to $\{0, 1\}$ is huge–much, much larger than the set of all Turing machines since we have assumed that they can be enumerated. Thus, at a high level the idea is the same as Cantors' but here we are showing that there is no bijective mapping from the set of all TM's to the set of all such functions.

But we said that inputs should be single numbers and not pairs of numbers! To handle this, we can encode a pair of numbers (i, j) by the single number $2^i \times 3^j$. Check that i and j can be uniquely extracted from $2^i \times 3^j$.

To show that $\mathcal{H}$ is not computable by any TM, we will introduce a second idea called a *reduction* that is extremely powerful and used extensively in Computer Science. In fact, you've seen this idea already although it didn't have this fancy name – it is none other than a proof by contradiction. Say that we want to prove $\neg A$, and we already know $\neg B$. Suppose that we can prove $A \Rightarrow B$. Then assume for sake of contradiction that A is true, thus by modus ponens it follows that B is true, which contradicts $\neg B$. To instantiate this in our setting, we let B be the statement that $\mathcal{D}$ is computable by any TM, and let A be the statement that $\mathcal{H}$ is computable by a TM. Since we have already proven $\neg B$, it is just left to prove $A \Rightarrow B$; that is, we want to prove that if $\mathcal{H}$ is computable by a TM, then $\mathcal{B}$ is also computable by a TM, in order to get a contradiction and therefore conclude that $\mathcal{H}$ is not computable. For this choise of A and B, proving $A \Rightarrow B$ is called a reduction (from B to A) because we are showing that computing B essentially reduces to the task of computing A.

So our remaining task is therefore to show that if we can compute $\mathcal{H}$,

then we can compute $\mathcal{D}$ by a TM. Here we will have to wave our hands a little bit, since we haven't even formally defined Turing machines! But we did say that they satisfy property (1), and thus we will argue informally that if we have an algorithm for $\mathcal{H}$, then we can also construct an algorithm for $\mathcal{D}$. How would we compute $\mathcal{D}$ in the first place? Remember that the input is a number i, and we want to determine of M_i halts and accepts i. The first step on input i is to actually find the TM program M_i. This can be carried out by enumerating all TMs until we get to the i^{th} one.

This is very inefficient but it will suffice for our purposes here. There are much more efficient ways to do this.

Now that we have M_i, how can we tell if M_i accepts i? If we just simulate M_i on input i, we may run into a problem if M_i doesn't halt on i since in that case our simulation will run forever and we will never know when to stop the simulation and output 1. But we are saved by the fact that we are assuming that we have an algorithm for $\mathcal{H}$!! Thus we can *first* run the algorithm for $\mathcal{H}$ on the input pair (i, i). If it accepts, then we know that M_i halts on i, so in this case we can go ahead and simulate M_i on i, and return the opposite answer. If on the other hand the algorithm for $\mathcal{H}$ on (i, i) rejects, then we know that M_i does not halt on i, so we should just return 1 (and not bother to do the simulation). Thus informally we have argued that if $\mathcal{H}$ is computed by some TM, then $\mathcal{D}$ is also computed by some TM, so we can conclude that $\mathcal{H}$ is not computable!

Other undecidable problems

Using this idea of a reduction, we can now prove that many other problems of interest are also not computable by any Turing machine. One of the most famous of these problems is called Hilbert's Tenth Problem. In 1900, the Second International Congress of Mathematicians was held in Paris, France, where David Hilbert, one of the greatest mathematicians in the world, was invited to deliver one of the main lectures. His lecture has become very famous because in his lecture, entitled "Mathematical Problems," he formulated 23 major mathematical problems that he felt were the most important open problems in all of mathematics to be studied in the coming century. Several of them have turned out to be very influential for mathematics of the 20th century. Some famous examples are: determining the truth or falsity of the continuum hypothesis, the Reimann hypothesis, formulating the axioms of physics, and proving that the axioms of arithmetic are consistent.

One of the most important is his tenth problem, called "Determining the solvability of a Diophantine equation" and asks, given a polynomial equation with any number of variables and integer coefficients, to

devise an algorithm to determine whether the equation has an integer solution. This was open for a very long time until in 1970 Yuri Matiyasevich finally resolved Hilbert's tenth problem by proving that it has no solution since it is *undecidable*! The proof is a *complicated* reduction using insights from Julia Robinson, and a connection to Fibonacci numbers.

Godel's legacy: the limitations of proofs

Another very famous problem that is not computable is called the Entscheidungsproblem. Informally, this is the problem of determining whether or not a mathematical statement is valid. We start with a fixed set of axioms (such as the axioms of Peano arithmetic, the most standard set of axioms for reasoning in number theory). The input is a mathematical sentence s, and the output should be 0 (reject) if s is not a logical consequence of the axioms, and 1 (accept) if s is a logical consequence of the axioms. This problem is undecidable by Matiyasevich's theorem, since the existence of solutions for Diophantine equations are a special type of mathematical statement. However, it is also possible give a simpler reduction showing that the Entscheidung problem is undecidable. Philosophically this is quite interesting as it proves that mathematics cannot be fully automated.

Entscheidung is the German word for "decision".

Closely connected to the Entscheidung problem is Hilbert's second problem, to prove that the axioms of arithmetic are consistent. In 1931 Kurt Godel proved his famous incompleteness theorems, essentially showing that there is no reasonable set of axioms that can capture all sentences that are true about the natural numbers. While his proof did not mention anything about computers or computability, we now know that his theorems are in fact very closely connected to undecidability, and can be proven using the ideas of reductions.

By "capture" we mean that the set of sentences that are logically consequences of the axioms should be exactly those sentences that are true over the natural numbers.

P versus NP

In the 60's and 70's the complexity class P emerged. It captures those decision problems that can be computed efficiently – where the number of basic computation steps in order to arrive at the answer is at most polynomial in the input length. That is, the runtime is $n^{O(1)}$. There are many examples of important problems in $\mathcal{P}$ and you will study them in many of your courses. For example all of these problems have polynomial-time algorithms: detecting whether a graph contains a cycle, determining whether a graph contains a perfect matching, and computing the greatest common divisor of two numbers. A larger class of

decision problems is known as $\mathcal{NP}$ and contains important problems such as whether a graph contains a clique of size $n/2$, and whether there is a boolean assignment to the variables of a propositional formula forcing it to true. $\mathcal{NP}$-complete problems are the hardest problems in the class $\mathcal{NP}$ and the best algorithms for these problems run in time that is *exponential* in n – that is, in time $2^{O(n)}$. The class *NP* is very important because it contains many many important problems that range across all disciplines, including fundamental problems in computational biology, physics, machine learning, and of course computer science. For all of these problems, all known algorithms run in exponential time, which makes them completely infeasible to solve. On the other hand, it is *not* known if it is possible to solve these problems much more efficiently, say in polynomial time. The P versus NP problem is the open problem of whether or not any of the $\mathcal{NP}$-complete problems can be solved in polynomial time, and is one of the most important open problems in mathematics and computer science today.

$\mathcal{NP}$ stands for nondeterministic polynomial time

It turns out that if one can get a polynomial-time algorithm for *any* NP-complete problem, then all problems in NP also have efficient algorithms. This was proven by Cook and independently by Levin in the early 70's.

Other cool applications: Cryptography

As we mentioned in the introduction to these notes, cryptography is the study of algorithms and protocols for doing cool things across the internet in the presence of adversaries. The techniques and tools that have been developed in cryptography are often very surprising and incredibly creative. Cryptography is in some sence the flip side of complexity theory. Whereas lower bounds in complexity theory prove that certain problems are inherently hard in that they require an infeasible amount of time in order to solve, cryptography *uses* this hardness in order to develop protocols! That is, who are these adversaries anyway? They are people or other computers, and thus they are limited to performing polynomial-time computation. In crytography, the computational hardness of problems is used to an advantage — to build protocols for various tasks, where the security of the protocols can be *proven* under the assumptions that the adversaries are polynomially-bounded, and that certain problems in complexity theory are infeasible.