Course notes for CSC 165 H: Mathematical Expression and Reasoning for Computer Science

Fall 2014

Gary Baumgartner *Danny Heap* *Richard Krueger* *François Pitt*

Department of Computer Science
University of Toronto

Contents

Chapter 1

Introduction

1.1 What's csc 165 h about?

In addition to hacking, computer scientists have to be able to understand program specifications, APIs, and their workmate's code. They also have to be able to write clear, concise documentation for others.
In our course you'll work on:

- Expressing yourself clearly (using English and mathematical expression).
- Understanding technical documents and logical expressions.
- Deriving conclusions from logical arguments, several proof techniques.
- Analyzing program efficiency.

In this course we care about communicating precisely:

- Knowing and saying what you mean.
- Understanding what others say and mean.

We want this course to help you during your university career whenever you need to read and understand technical material — course textbooks, assignment specifications, *etc.*

Who needs csc 165 h?

You need this course if you do:

> memorize math;
>
> have trouble explaining what you are doing in a mathematical or technical question;
>
> have trouble understanding word problems.

You need this course if you don't:

> like reading math textbooks to learn new math;
>
> enjoy talking about abstract x and y just as much as when concrete examples are given for x and y;
>
> have a credit for csc 238 h in your academic history, or intend to take csc 240 h.

Why does CS need mathematical expressions and reasoning?

We all enjoy hacking, that is designing and implementing interesting algorithms on computers. Perhaps not all of us associate this with the sort of abstract thinking and manipulation of symbols associated with mathematics. However there is a useful two-way contamination between mathematics and computer science. Here are some examples of branches of mathematics that taint particular branches of Computer Science:

Computer graphics use multi-variable calculus, projective geometry, linear algebra, physics-based modelling

Numerical analysis uses multivariable calculus and linear algebra

Cryptography uses number theory, field theory

Networking uses graph theory, statistics

Algorithms use combinatorics, probability, set theory

Databases use set theory, logic

AI uses set theory, probability, logic

Programming languages use set theory, logic

How to do well in csc 165 h

Check the course web page and the course forum frequently.

Understand the course information sheet. This is the document that we are committed to live by in this course.

Get in the habit of asking questions and contributing to the answers.

Spend time on this course. The model that we instructors assume is that you work an average of 8–10 hours per week on this course (3–4 in lecture and tutorial, and 5–6 reviewing notes, working on assigned problems, attending office hours (as required), *etc.*) Any material that's new to you will require time for you to really acquire it and use it on other courses.

Don't plagiarize. Passing off someone else's work as your own is an academic offense. Always give generous and complete credit when you consult other sources (books, web pages, other students).

About these notes

These notes were originally created by Gary Baumgartner (and contributed to by many others), expanded and typeset by Danny Heap and Richard Krueger, and further expanded and modified by François Pitt. These notes are written to stand alone and cover the material included in the present csc 165 h syllabus without the need of a supplementary textbook (though it's often advantageous to read another perspective). Please let us know of any typos or errors, or anything that seems (unintentionally) confusing on the first read, so we can make appropriate corrections.

In these notes you'll find numerous superscripts.[1] These often indicate answers to questions worked out in lecture, and through the wonders of word processing, those answers are formatted as endnotes (at the end of the chapter). Our motivation isn't so much to give you whiplash moving your gaze between the question and the answer, as to allow you to form your own answer before looking at our version.

1.2 Human versus technical communication

Natural languages (English, Chinese, Arabic, for example) are rich and full of potential ambiguity. In many cases humans speaking these languages share a lot of history, context, and assumptions that remove or reduce the ambiguity. If we don't share (or choose to momentarily forget) the history and context, there is a rich source of humour in double-meanings created by natural languages. For example, consider these headlines listed on http://www.departments.bucknell.edu/linguistics/semhead.html:

> Prostitutes appeal to Pope
>
> Iraqi head seeks arms
>
> Police begin campaign to run down jay-walkers
>
> Death may cause loneliness, feelings of isolation
>
> Two sisters reunite after 18 years at checkout counter.[2]

Computers are notorious for lacking a sense of humour, and we communicate with them using extremely constrained languages called programming languages. In programming languages, expressions aren't expected to be ambiguous.

Human technical communication about computing must be similarly constrained. We have to assume less common history and context is shared with the other humans participating in technical communication, and misplaced assumptions can result in catastrophe. We aim for increased PRECISION, that is a smaller tolerance for ambiguity. We will use MATHEMATICAL LOGIC, a precise language, as a form of communication in this course.

Mathematicians share a common dialect to talk unambiguously about particular concepts in their work (*e.g.*, "differentiable functions are continuous"). Often ordinary words ("continuous") are used with restricted, or special, meanings. The same word may have different technical meaning in different mathematical contexts, for example GROUP may mean one thing in group theory, another in combinatorial design theory.

Since technical language is used between human beings, some degree of ambiguity is tolerated, and probably necessary. For example, an audience of Java programmers would not object to the subtle shift in the meaning used for "a" in the following fragments:[3]

```
/** Sorts a in ascending order */

public void sort(int[] a) ...
```

versus

```
// sets a to 1

a = 1;
```

Since another human is reading our comments, this potential for double meaning is benign. A computer reading our comments would, of course, be unforgiving. However, Java programmers have to assume familiarity with programming from their audience, to avoiding driving others crazy by writing long comments (and being driven crazy by long comments written by others).

In this course we can't assume the context necessary to always conclude that you know what you're saying, so you'll have to demonstrate it explicitly. On the other hand, you will learn to understand somewhat imprecise statements that can be made precise from the context.

1.3 PROBLEM-SOLVING

Of course, as a computer scientist you are expected to do more than express yourself clearly about the algorithms, methods, and classes that you either develop or use. You must also work on solving new and challenging problems, sometimes without even knowing in advance whether a solution is possible. You will learn to balance insights that may not be fully articulate, with rigour that convinces yourself and others that your insights are correct. You need both pieces to succeed.

We will try to teach some techniques that increase your chances of gaining insight into mathematical problems that you encounter for the first time. Although these techniques aren't guaranteed to succeed for every mathematical problem, they work often enough to be useful.

Much of our approach is based on George Polya's in "How to Solve it," and other books following that approach. Although you can find lots of references to this on the web, here's a précis of Polya's approach:

UNDERSTAND THE PROBLEM: Make sure you know what is being asked, and what information you've been given. It helps to re-state the problem (sometimes several times) in your own words, perhaps representing it in different ways or drawing some diagrams.

PLAN A SOLUTION: Perhaps you've seen a similar problem. You might be able to use either its result, or the method of solving it. Try working backwards: assume you've solved the problem and try to deduce the next-to-last step in solving it. Try solving a simpler version of the problem, perhaps solving small or particular cases.

CARRY OUT YOUR PLAN: See whether your plan for a solution leads somewhere. It may be necessary to repeat parts of the earlier steps. When you're stuck, try to articulate exactly what you're missing.

REVIEW ANY SOLUTION YOU ACHIEVE: Look back on any pieces of the puzzle you solve, try to remember what lead to breakthroughs and what blocked progress. Carefully test your solution until you're convinced (and can convince a skeptical peer) that you've got a solution. Extend the solved problem to new problems.

Notice how these steps differ from our usual pattern of either avoiding work on a problem (staring at a blank page), or diving in without a plan. The very idea of separating the plan for finding a solution from the act of finding a solution seems weird and unnatural. We'll add a further unnatural suggestion: you should keep a record (notes or a journal) of your problem-solving attempts. This turns out to be useful both in solving the problem at hand and later, related problems.

Here's an example of a real-life problem (eavesdropping on a streetcar) that you might apply Polya's approach to. You're swinging from the grip on a streetcar during rush hour, and you hear the following conversation fragments behind you, between persons A and B:

PERSON A: I haven't seen you in ages! How old are your three kids now?

PERSON B: The product of their ages (in years) is 36. [You begin to suspect that B is a difficult conversation partner].

PERSON A: That doesn't really answer my question...

PERSON B: Well, the sum of their ages (in years) is — [at this point a fire engine goes by and obscures the rest of the answer].

PERSON A: That still doesn't really tell me how old they are.

PERSON B: Well, the eldest plays piano.

PERSON A: Okay, I see, so their ages are — [at this point you have to get off, and you miss the answer].

1.4 Inspirational puzzles

As inspiration to the usefulness of mathematical logic and reasoning to solving problems, we submit to you the following puzzles. Each is related to problems common in computer science, and is interesting in its own right. The difficulty of these puzzles varies widely, and we intentionally give no indication of their presumed difficulty nor their solutions. By the end of this course, you will likely be able to solve most of these puzzles (indeed many you may be able to solve now).

- 3 boxes
  Suppose you are a contestant on a game show and you are presented with 3 boxes. Inside one is a prize, which you will win if you chose the correct box. The game goes thusly: you choose a box, and the host opens one of the remaining boxes which is empty. You may then switch your choice to the remaining box or stay with your original choice. Which box would you choose for the best chance of winning the prize? Why?

- 3 labelled boxes
  In the next round, you are again presented with 3 boxes, one containing a small prize. This time, you must choose one box, and if you choose correctly, you win the prize. On closer examination, you notice a label on each box.

The prize is not here.	The prize is here.	The prize is not here.

  Which box do you choose?

- 2 labelled boxes
  In the next round, you are presented with just two boxes, with one containing a prize. The host explains that two stagehands, Adam and Brian, pack the boxes. Adam always puts a true statement on the box, and Brian always puts a false statement on the box. You don't know who packed the boxes, or even if they were both packed by the same person or different people. The boxes say:

The prize is not here.	Exactly one box was packed by Brian.

  Which box do you choose?

- 2 labelled boxes surprise
  In the final round, you are presented with two more boxes. The host tells you one box contains the grand prize, but if you choose the wrong one, you lose everything. The labels say:

The prize is not here.	Exactly one statement on the boxes is false.

  You reason that if the statement on the right box is true, the left box statement must be false, so the prize is in the left box. If the statement on the right box is false, either both statements are true or both are false. They cannot both be true, since the one on the right is false. So both are false, and the prize must be in the left box. So you choose the left box.

  You open it and it's empty! The host claims he didn't lie to you. What's wrong? (The difference between this and the previous puzzle is subtle but basic and essential for rigorous treatment of logic.)

- Knights and Knaves
  On the island of knights and knaves, every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You come across two inhabitants, let's call them A and B.

  1. Person A says: "I am a knave or B is a knight." Can you determine what A and B are?
  2. Person A says: "We are both knaves." What are A and B?
  3. Person A says: "If B is a knave, then I'm a knight." Person B says: "We are different." What are A and B?

4. You ask A: "Are you both knights?" A answers either Yes or No, but you don't know enough to solve the problem. You then ask B: "Are you both knaves?" B answers either Yes or No, and now you know the answer. What are A and B?
5. A and B are guarding two doors, one leading to treasure and one leading to a ferocious lion which will surely eat you. You must choose one door. You may ask one guard one yes/no question before choosing a door. What question do you ask? Is it easier if you know one is a knight and one is a knave (but you don't know which is which)?

- Mother Eve
  Theorem: There is a woman on Earth such that if she becomes sterile, the whole human race will die out because all women will become sterile.

  Proof: Either all women will become sterile or not. If yes, then any woman satisfies the theorem. If no, then there is some woman that does not become sterile. She is then the one such that if she becomes sterile (but she does not), the whole human race will die out.

  Is this argument convincing?

- Santa Claus
  Wife: Santa Claus exists, if I am not mistaken.
  Husband: Well of course Santa Claus exists, if you are not mistaken.
  Wife: Hence my statement is true.
  Husband: Of course!
  Wife: So I was not mistaken, and you admitted that if I was not mistaken, then Santa Claus exists. Therefore Santa Claus exists.

  Are you convinced? Why or why not?

- Daemon in a Pentagon
  There is a pentagon and at each vertex there is an integer number. The numbers can be negative, but their sum is positive. A daemon living inside the pentagon manipulates the numbers with the following atomic action. If it spots a negative number at one of the vertices, it adds that number to its two neighbours and negates the number at the original vertex. Prove that no matter what numbers we start with, eventually the daemon cannot change any of the numbers.

1.5 Some mathematical prerequisites

Here are some mathematical concepts, and notation, that we'll assume you are comfortable with during this course. We won't necessarily be teaching this material, so the onus is on you to make sure you really are comfortable with this material and if not, to ask about it.

You may also want to refer to this section when justifying conclusions in proofs you write up for this course.

Set theory and notation

A set is a collection of 0 or more "things". These things are called elements of the set and are often presented as a list surrounded by curly brackets (braces), with a comma between each element.

$\mathbb{Z}$: The integers, or whole numbers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

$\mathbb{N}$: The natural numbers or non-negative integers $\{0, 1, 2, \ldots\}$. Notice that the convention in Computer Science is to include 0 in the natural numbers, unlike in some other disciplines.

$\mathbb{Z}^+$: The positive integers $\{1, 2, 3, \ldots\}$.

$\mathbb{Z}^-$: The negative integers $\{-1, -2, -3, \dots\}$.

$\mathbb{Z}^*$: The non-zero integers $\{\dots, -2, -1, 1, 2, \dots\} = \mathbb{Z} - \{0\} = \mathbb{Z}^- \cup \mathbb{Z}^+$.

$\mathbb{Q}$: The rational numbers (ratios of integers), comprised of $\{0\}$, $\mathbb{Q}^+$ (positive rationals), and $\mathbb{Q}^-$ (negative rationals). The set of all numbers of the form p/q, where $p \in \mathbb{Z}$ and $q \in \mathbb{Z}^*$.

$\mathbb{R}$: The real numbers, comprised of $\{0\}$, $\mathbb{R}^+$ (positive reals), and $\mathbb{R}^-$ (negative reals). The set of all numbers of the form $m.d_1d_2d_3\cdots$, where $m \in \mathbb{Z}$ and $d_1, d_2, d_3, \dots \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

$x \in A$: "x is an element of A," or "x is in A."

$A \subseteq B$: "A is a subset of B." Every element of A is also an element of B.

$A = B$: "A equals B." A and B contain exactly the same elements, in other words $A \subseteq B$ and $B \subseteq A$.

$A \cup B$: "A union B." The set of elements that are in either A, or B, or both.

$A \cap B$: "A intersection B." The set of elements that are in both A and B.

$A \setminus B$ or $A - B$: "A minus B." The set of elements that are in A but not in B (the set difference).

$|A|$: "cardinality of A." The number of elements in A.

$\varnothing$ or $\{\}$: "The empty set." A set that contains no elements. By convention, for *any* set A, $\varnothing \subseteq A$ (we will see a logical justification for this fact when we discuss vacuous truth in Section 2.7).

$\mathcal{P}(A)$: "The power set of A." The set of all subsets of A. For example, suppose $A = \{73, \diamond\}$, then $\mathcal{P}(A) = \{\varnothing, \{73\}, \{\diamond\}, \{73, \diamond\}\}$.

$\{x : P(x)\}$ or $\{x \mid P(x)\}$: "The set of all x for which $P(x)$ is true." For example, $\{x \in \mathbb{Z} : \cos(\pi x) > 0\} = \{\dots, -4, -2, 0, 2, 4, \dots\}$ (even integers).

Number theory

If m and n are natural numbers, with $n \neq 0$, then there is exactly one pair of natural numbers (q, r) such that:

$$m = qn + r, \qquad n > r \geq 0.$$

We say that q is the quotient of m divided by n, and r is the remainder. We also say that $m \bmod n = r$.

In the special case where the remainder r is zero (so $m = qn$) we say that n divides m and write $n \mid m$. We say that n is a divisor of m (*e.g.*, 4 is a divisor of 12). Convince yourself that any natural number is a divisor of 0, and that 1 is a divisor of any natural number.

A natural number, p, is prime if it has exactly two positive divisors. Thus $2, 3, 5, 7, 11$ are all prime but 1 is not (too few positive divisors) and 9 is not (too many positive divisors). There are infinitely many primes, and any integer greater than 1 can be expressed (in exactly one way) as a product of one or more primes.

Functions

We'll use the standard notation $f : A \to B$ to say that f is a function from set A to B. In other words, for every $x \in A$ there is an associated $f(x) \in B$. Here are some common number-theoretic functions along with their properties. We'll use the convention that variables $x, y \in \mathbb{R}$ whereas $m, n \in \mathbb{Z}^+$.

$\min\{x, y\}$: "minimum of x or y." The smaller of x or y. Properties: $\min\{x, y\} \leq x$ and $\min\{x, y\} \leq y$.

$\max\{x, y\}$: "maximum of x or y." The larger of x or y. Properties: $x \leq \max\{x, y\}$ and $y \leq \max\{x, y\}$.

$|x|$: "absolute value of x," which is $\begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$
Notice that similar notation is used for the cardinality of a set, so you have to pay attention to the context.

$\gcd(m, n)$: "greatest common divisor of m and n." The largest positive integer that divides both m and n.

$\text{lcm}(m, n)$: "least common multiple of m and n." The smallest positive integer that is a multiple of both m and n. Property: $\gcd(m, n) \times \text{lcm}(m, n) = mn$.

$\lfloor x \rfloor$ OR $\text{floor}(x)$: The largest integer that is not larger than x,

$$\forall x \in \mathbb{R}, y = \lfloor x \rfloor \Leftrightarrow y \in \mathbb{Z} \wedge y \leq x \wedge (\forall z \in \mathbb{Z}, z \leq x \Rightarrow z \leq y)$$

$\lceil x \rceil$ OR $\text{ceil}(x)$: The smallest integer that is not smaller than x,

$$\forall x \in \mathbb{R}, y = \lceil x \rceil \Leftrightarrow y \in \mathbb{Z} \wedge y \geq x \wedge (\forall z \in \mathbb{Z}, z \geq x \Rightarrow z \geq y)$$

INEQUALITIES

FOR ANY $m, n \in \mathbb{Z}$: $m < n$ if and only if $m + 1 \leq n$, and $m > n$ if and only if $m \geq n + 1$.

FOR ANY $x, y, z, w \in \mathbb{R}$:

- If $x < y$ and $w \leq z$, then $x + w < y + z$.
- If $x < y$, then $\begin{cases} xz < yz & \text{if } z > 0 \\ xz = yz & \text{if } z = 0 \\ xz > yz & \text{if } z < 0 \end{cases}$
- If $x < y$ and $y \leq z$ (or $x \leq y$ and $y < z$), then $x < z$.
- $|x + y| \leq |x| + |y|$. This is an example of the TRIANGLE INEQUALITY.

EXPONENTS AND LOGARITHMS

FOR ANY $a, b, c \in \mathbb{R}^+$: $a = \log_b c$ if and only if $b^a = c$.

FOR ANY $x \in \mathbb{R}^+$: $\ln x = \log_e x$ and $\lg x = \log_2 x$

FOR ANY $a, b, c \in \mathbb{R}^+$ AND $n \in \mathbb{Z}^+$:

$$\sqrt[n]{b} = b^{1/n} \qquad b^{\log_b a} = a = \log_b b^a$$
$$b^a b^c = b^{a+c} \qquad log_b(ac) = \log_b a + \log_b c$$
$$(b^a)^c = b^{ac} \qquad \log_b(a^c) = c \log_b a$$
$$b^a / b^c = b^{a-c} \qquad \log_b(a/c) = \log_b a - \log_b c$$
$$b^0 = 1 \qquad log_b 1 = 0$$
$$a^c b^c = (ab)^c$$

Chapter 1 Notes

[1]Like this.

[2]The word "appeal" in the first headline has two meanings, so one interpretation is that the Pope is fond of prostitutes, and another is that prostitutes have asked the Pope for something. The words "head" and "arms" each have two meanings, so one interpretation is that the body part above some Iraqi person's shoulders is looking for the appendages below their shoulders, and another is that the most senior Iraqi is looking for weapons. The phrase "run down" can mean either hitting with a car or looking for. In the fourth headline, it's not clear to whom death causes loneliness and isolation: the dead person or their survivors. In the fifth headline it's not clear whether the checkout line was moving REALLY slowly, or that the checkout counter was just the location of their reunion.

[3]In the first fragment "a" means "the object referred to by the value in a." In the second "a" means "the variable a."

Chapter 2

Logical Notation

2.1 Universal quantification

Consider the following table that associates employees with properties:

Employee	Gender	Salary
Al	male	60,000
Betty	female	500
Carlos	male	40,000
Doug	male	30,000
Ellen	female	50,000
Flo	female	20,000

Claims about individual objects can be evaluated immediately (Al is male, Flo makes 20,000). But the tabular form also allows claims about the entire database to be considered. For example:

> Every employee makes less than 70,000.

Is this claim true? So long as we restrict our universe to the six employees, we can determine the answer.[1] When a claim is made about all the objects (in this context, humans are objects!) being considered (*i.e.*, in our "universe"), this is called Universal Quantification. The meaning is that we make explicit the logical quantity (we "quantify") every member of a class or universe. English being the slippery object it is allows several ways to say the same thing:

> Each employee makes less than 70,000.
>
> All employees make less than 70,000.
>
> Employees make less than 70,000.[2]

Our universe (aka "domain") is the given set of six employees. When we say every, we mean every. This is not always true in English, for example "Every day I have homework," probably doesn't consider the days preceding your birth or after your death. Now consider:

> Each employee makes at least 10,000.

Is this claim true? How do you know?[3] A single counter-example is sufficient to refute a universally-quantified claim. What about the following claim:

> All female employees make less than 55,000.

Is this claim true? Restrict the domain and check each case.[4] What about

Every employee that earns less than 55,000 is female?[5]

How about this claim:

Every male employee makes less than 55,000.

It worked for females.[6] Notice a pattern. To disprove a universally-quantified statement you need just one counter-example. To prove one you need to consider every element in a domain. A universally-quantified statement of the form

Every P is a Q

needs a single COUNTER-EXAMPLE to disprove, and verification that every element of the domain is an EXAMPLE to prove.

2.2 Existential quantification

Here's another sort of claim:

Some employee earns over 57,000.

At first this claim doesn't seem to be about the whole database, but just about an employee who earns over 57,000 (if that employee exists, and Al does exist). But what about:

There is an employee who earns less than 57,000.

This claim is also true, and it is verified by any of the employees in the set {Betty, Carlos, Doug, Ellen, Flo}. It's not a claim about any particular employee in that five-member set, but rather a claim that the set isn't empty. Although the non-empty set might have many members, one example of a member of the set is enough to show that it's not empty. Now consider:

Some employee earns over 80,000.

This claim is false. There isn't an employee in the database who earns over 80,000. To show the set of employees earning over 80,000 is empty, you have to consider every employee in the universe and demonstrate that they don't earn over 80,000.
In everyday language existential quantification is expressed as:

There [is / exists] [a / an / some / at least one] ... [such that / for which] ...,
or [For] [a / an / some / at least one] ..., ...

Note that the English word "some" is always used INCLUSIVELY here, so "some object is a P" is true if every object is a P.

The claims are about the EXISTENCE of one or more elements of a domain with some property, and they are examples of EXISTENTIAL quantification. Existential quantification requires you to exhibit just one EXAMPLE of an element with the property to prove, but it requires you to consider the entire domain to show that every element is a COUNTER-EXAMPLE to disprove.

The anti-symmetry between universal and existential quantification may be better understood by switching our point of view from properties to the sets of elements having those properties.

2.3 Properties, sets, and quantification

Let's look at that table again.

Employee	Gender	Salary
Al	male	60,000
Betty	female	500
Carlos	male	40,000
Doug	male	30,000
Ellen	female	50,000
Flo	female	20,000

Saying that Al is male is equivalent to saying Al belongs to the set of males. Symbolically we might write Al $\in M$ or M(Al). It's useful and natural to interchange the ideas of properties and sets. If we denote the set of employees as E, the set of female employees as F, the set of male employees as M, and the set of employees who earn less than 55,000 as L, then we have a notation for concisely (and precisely) evaluating claims such as M(Flo),[7] or L(Carlos).[8] So far the notation doesn't seem to have achieved much, but how about:

> Everything in F is also in L (in other notation, $F \subseteq L$)?

So our universally-quantified claim that all females make less than 55,000 turns into a claim about subsets. We already have some intuition about subsets, so let's put it to work by drawing a Venn diagram (see Figure 2.1). Make sure you are solid on the meaning of "subset." Is a set always a subset of itself?[9] Is the empty set (the set with no elements) a subset of any set?[10]

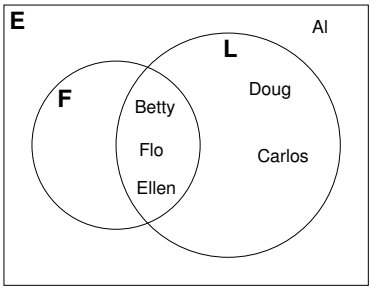

Figure 2.1: The only elements of F are also elements of L, so $F \subseteq L$. In this particular diagram, the maximum number of regions consistent with $F \subseteq L$ are occupied: three out of the four regions are occupied.

Now consider the claim

> Something in M is also in $\overline{L}$: there is some male who does not earn less than 55,000

The complement of L is sometimes denoted $\overline{L}$, and means elements that are not in L. One way to denote "something in M is also in $\overline{L}$" in set notation is $M \cap \overline{L} \neq \varnothing$ — saying "something" is in both sets is the same as saying their intersection is non-empty. Now, you should be able to compare this to the definition of a subset to see that this is same as saying that M is not a subset of L, or $M \not\subseteq L$.
The anti-symmetry of universal and existential quantification becomes systematic:

- Every P is a Q means $P \subseteq Q$. To prove this claim you need to consider every element of P and show they are also elements of Q. To disprove this claim, you need to find just one element of P that is not an element of Q.

- Some P is a Q means $P \not\subseteq \overline{Q}$. To prove this you need to find just one P that isn't a non-Q (a roundabout way of saying find just one P that is a Q). To disprove it, you must consider every P and show they are also non-Qs.

2.4 SENTENCES, STATEMENTS, AND PREDICATES

Recall the table of employees with their genders and salaries from above:

EMPLOYEE	GENDER	SALARY
Al	male	60,000
Betty	female	500
Carlos	male	40,000
Doug	male	30,000
Ellen	female	50,000
Flo	female	20,000

Now consider the following claims:

CLAIM 2.1: The employee makes less than 55,000.

CLAIM 2.2: Every employee makes less than 55,000.

Can you decide whether both claims are true or false?[11] The basic difference between the two claims is that Claim 2.1 is about a particular employee, and it is true or false depending on the earnings of that employee, whereas Claim 2.2 is about the entire set of employees, E, and it is true or false depending on where that set of employees stands in relation to the set L, those who earn over 55,000.

Claim 2.1 is called a SENTENCE. It may refer to unquantified objects (for example "the employee"). Once the objects are specified (substitutions are made for the variable(s)), a sentence is either true or false (but never both). Claim 2.2 is called a STATEMENT. It doesn't refer to any unquantified variables, and it is either true or false (never both). Every statement is a sentence, but not every sentence is a statement. If you want to make it explicit that a sentence refers to unquantified objects, you may call it an "open sentence." Thus a sentence is a statement if and only if it is not open. Universal quantification transformed Claim 2.1 into Claim 2.2, from an open sentence about an unspecified element of the set of employees, into a statement about the (specified in the database) sets of Employees and those earning over 55,000.

SYMBOLS

Symbols are useful when they make expressions clearer and highlight patterns in similar expressions. We already moved in the direction of making our logical expressions symbolic by naming sets E (employees), F (females), and L (those earning less than 55,000). Naming gives us a concise expression for these sets, and it emphasizes the similar roles these sets play. We introduce more symbolism into our sentences, statements, and predicates now.

As a programmer you create a sentence every time you define a boolean function. In logic, a predicate is a boolean function. For convenience you can name your predicate, and you can define it by showing how it evaluates its input, using a symbol to stand for generic input. For example, if L is the set of employees earning less than 55,000

$$L(x)\text{: } x \in L.$$

Notice how similar this is to defining a function in a programming language in terms of how it evaluates its parameters. The symbol x is useful in the definition — it holds the parentheses, "(" and ")", apart so that we can see that exactly one value is needed, and it shows where to plug that value into the definition. Notice that this definition would mean the same things if we replaced the symbol x with the symbol y or the symbol y^3. The symbol x doesn't specify any value that helps determine whether our predicate evaluates to true or false. Our open sentence above, Claim 2.1, is equivalent to $L(x)$ — we can't evaluate it without substituting something from the set E for x. L(Carlos) is true, L(Al) is false.

Claim 2.2 is equivalent to "for all employees x, $L(x)$." The phrase "for all employees x" quantifies the variable x, and changes the claim from an open sentence about unspecified x to a statement about sets E and L, which were specified in the database above. Of course, in this context, "employees" refers to those in our database, and not any other employees.

We can indicate universal quantification symbolically as $\forall$, read as "for all." This makes sense if we specify the universe (domain) from which we are considering "all" objects. With this notation, Claim 2.2 can be written

$\forall$ employees, the employee makes less than 55,000.

Things become clearer if we introduce a name for the unspecified employee:

$\forall$ employees x, x makes less than 55,000.

Since this statement may eventually be embedded in some larger and more complicated structure, we can add to the brevity and clarity by adding a bit more notation. Let E denote the set of employees, and $L(x)$ denote the predicate "x makes less than 55,000." Now Claim 2.2 becomes:

$\forall x \in E, L(x)$.

We can do something similar with existential quantification. We can transform $L(x)$ into a statement by saying there is some element of E that also belongs to L:

There exist employees who earn less than 55,000.

$\exists x \in E, L(x)$.

The symbol we use for "there exists" is $\exists$. This is a statement about the sets E and L (it says they have a common, non-empty subset), and not a statement about individual elements of those sets. The symbol x doesn't stand for a particular element, it rather indicates that there is at least one element common to E and L.

2.5 Implications

Consider a claim of the form

IF an employee is male, THEN he makes less than 55,000.

This is called an IMPLICATION. It says that for employees, being male IMPLIES making less than 55,000.[12] This is universal quantification in disguise, since it could be accurately re-expressed as "Every male employee earns less than 55,000," or $\forall x \in E \cap M, L(x)$. Notice that the implication "males implies less than 55,000" has the same effect as restricting the domain by intersecting E with M in the universally-quantified statement. However, it turns out to be convenient sometimes to keep the implication "male implies less than 55,000" separate from the domain. In this way, we can consider the implication as part of universes other than E (perhaps H, the set of humans, or $X = \{\text{Doug}, \text{Carlos}\}$). Separating the implication from the surrounding universe also means we don't have to define a set for each predicate, so we could have "$M(x)$ implies $L(x)$" without necessarily defining the sets M and L (although we could always come up with suitable definitions if we needed to).

Just as with universal quantification, the only way to disprove the implication "if P then Q" is to show an instance where P is true but Q is false. If, in every possible instance, we have either not-P or Q, then the implication "if P then Q" is true.

In the implication "if P then Q," we call P the ANTECEDENT (sometimes the ASSUMPTION), and Q the CONSEQUENT (sometimes the CONCLUSION).

Since logical implication borrows the English word "if," we need to reject some of the common English uses of "if" that we don't mean when "if" is used in logic. In logic "if...then" tells you nothing about

causality. "If it rained yesterday, then the sun rose today," is a true implication, but the (possible) rain didn't cause the (certain) rising of the sun. Also, when my mother told me "if you eat your vegetables, then you can have dessert," she also meant "otherwise you'll get no dessert." In ordinary English, my mother used "if... then" to mean "if and only if... then." In logic we use the more constrained meaning. We want "If P then Q" to mean "Every P is a Q."
What does "every P is a Q" tell us? In our database example:

CLAIM 2.3: If an employee is female, then she makes less than 55,000.

Claim 2.3 discusses three sets, E, the set of employees, F, the set of female employees, and L, the set of employees making less than 55,000. Claim 2.3 implicitly invokes universal quantification, so it is more than a claim about a particular employee. The Venn diagram Figure 2.1 indicates the situation corresponding to our table. If you had no access to either the table or the Venn diagram, but only knew the Claim 2.3 was true, what would you know about

1. F, the set of female employees? What else does the implication tell you about Ellen if you only know that Ellen is female?
2. L, the set of employees earning less than 55,000? What do you know about Betty (if you only know she's in L) or Carlos (if you only know he's in L)?
3. $\overline{F}$, the set of male employees? Think about both Doug and Al.
4. $\overline{L}$ (the complement of L), the set of employees making 55,000 or more.

Knowing "P implies Q" tells us nothing more about some sets,[13] however it does tell us more about others.[14] Suppose you have a new employee Grnflx (from a domain short of vowels), plus our Venn diagram (2.1). Which region of the Venn diagram would you add Grnflx to in order to make Claim 2.3 false?[15] Once that region is occupied, does it matter whether any of the other regions are occupied or not?[16]

MORE SYMBOLS

We can write implication symbolically as $\Rightarrow$, read "implies." Now "P implies Q" becomes $P \Rightarrow Q$. Claim 2.3 could now be re-written as

> an employee is female $\Rightarrow$ that employee makes less than 55,000.

CONTRAPOSITIVE

The CONTRAPOSITIVE of $P \Rightarrow Q$ is $\neg Q \Rightarrow \neg P$ ($\neg$ is the symbol for negation). In English the contrapositive of "all P is/are Q" is "all non-Q is/are non-P." Put another way, the contrapositive of "P implies Q" is "non-Q implies non-P." The contrapositive of Claim 2.3 is

> an employee doesn't make less than 55,000 $\Rightarrow$ that employee is not female.

or, given the structure of the domain E of employees:

> an employee makes at least 55,000 $\Rightarrow$ that employee is male.

Does the contrapositive of Claim 2.3 tell us everything that Claim 2.3 itself does? Check the Venn diagram (2.1). Does every Venn diagram that doesn't contradict Claim 2.3 also not contradict the contrapositive of Claim 2.3?[17] Can you apply the contrapositive twice? To do this it helps to know that applying negation ($\neg$) twice toggles the truth value twice (I'm not not going means I'm going). Thus the contrapositive of the contrapositive of $P \Rightarrow Q$ is the contrapositive of $\neg Q \Rightarrow \neg P$, which is $\neg\neg P \Rightarrow \neg\neg Q$, equivalent to $P \Rightarrow Q$.

Converse

The converse of $P \Rightarrow Q$ is $Q \Rightarrow P$. In words, the converse of "P implies Q" is "Q implies P." An implication and its converse don't mean the same thing. Consider the Venn diagram Figure 2.1. Would it work as a Venn diagram for $L \Rightarrow F$?[18]
Consider an example where the (implicit) domain is the set of pairs of numbers, perhaps $\mathbb{R} \times \mathbb{R}$.

Claim 2.4: $x = 1 \Rightarrow xy = y$

- If we know $x = 1$, then we know $xy = y$.
- If we know $x \neq 1$, then we don't know whether or not $xy = y$.
- If we know $xy = y$, then we don't know whether or not $x = 1$.
- If we know $xy \neq y$, then we know $x \neq 1$.

The contrapositive of Claim 2.4 is:

$$xy \neq y \Rightarrow x \neq 1.$$

Check the four points we knew from Claim 2.4, and see whether we know the same ones from the contrapositive (it may be helpful to read them in reverse order). What about the converse?

$$xy = y \Rightarrow x = 1$$

with equivalent contrapositive

$$x \neq 1 \Rightarrow xy \neq y.$$

The converse of Claim 2.4 is not equivalent to Claim 2.4, for example consider the pair $(5, 0)$, that is $x = 5$ and $y = 0$. Indeed, Claim 2.4 is true, while its converse is false.

Implication in everyday English

Here are some ways of saying "P implies Q" in everyday language. In each case, try to think about what is being quantified, and what predicates (or perhaps sets) correspond to P and Q.

- If P, [then] Q.
  "If nominated, I will not stand."
  "If you think I'm lying, then you're a liar!"
- When[ever] P, [then] Q.
  "Whenever I hear that song, I think about ice cream."
  "I get heartburn whenever I eat supper too late."
- P is sufficient/enough for Q
  "Differentiability is sufficient for continuity."
  "Matching fingerprints and a motive are enough for guilt."
- Can't have P without Q
  "There are no rights without responsibilities."
  "You can't stay enrolled in CSC 165 H without a pulse."
- P requires Q
  "Successful programming requires skill."

- For P to be true, Q must be true / needs to be true / is necessary
  "To pass CSC 165 H, a student needs to get 40% on the final."

- P only if / only when Q
  "I'll go only if you insist."

For the antecedent (P) look for "if," "when," "enough," "sufficient." For the consequent (Q) look for "then," "requires," "must," "need," "necessary," "only if," "when." In all cases, check whether the expected meaning in English matches the meaning of $P \Rightarrow Q$. In other words, you've got an implication if, in every possible instance, either P is false or Q is true.

2.6 Quantification and implication together

So far we have considered an implication to be universal quantification in disguise:

CLAIM 2.5: If an employee is male, then that employee makes less than 55,000.

The English indefinite article "an" signals that this means "Every male employee makes less than 55,000," and this closed sentence is either true or false, depending on the domain of employees. This can be expressed as $\forall x \in E, M(x) \Rightarrow L(x)$, and we can separate the "For all employees," portion from the "if the employee is male, then the employee makes less than 55,000," portion. Symbolically, we can think about $\forall x \in E$ separately from $M(x) \Rightarrow L(x)$, giving us some flexibility about which values we might substitute for x. This allows us to express the unquantified implication:

CLAIM 2.6: If the employee is male, then that employee makes less than 55,000.

The English definite article "the" often signals an unspecified value, and hence an open sentence. We could transform Claim 2.6 back into Claim 2.5 by prefixing it with "For every employee, ..."

CLAIM 2.7: For every employee, if the employee is male, then that employee makes less than 55,000.

Since the claim is about male employees, we are tempted to say $\forall m \in M, L(m)$, which would be correct if the only males we were considering were those in E — $\forall m \in E \cap M, L(m)$ would certainly capture what we mean. Using that approach we would restrict the domain that we are universally quantifying over by intersecting with other domains. However, it is often convenient to restrict in another way: set our domain to the largest universe in which the predicates make sense, and use implication to restrict further. We don't have to avoid reasoning about non-males when we say $\forall e \in E, M(e) \Rightarrow L(e)$, and we get the same meaning as $\forall m \in E \cap M, L(m)$.

It also often happens that the predicate expressed by $M(e)$ doesn't neatly translate into a set that can be intersected with set E, so the universally quantified implication format can be handy. For example, $\forall n \in \mathbb{N}, n > 0 \Rightarrow 1/n \in \mathbb{R}$ means the same things as $\forall n \in \mathbb{N} \setminus \{0\}, 1/n \in \mathbb{R}$, but expressing the set $\mathbb{N} \setminus \{0\}$ seems more awkward than using universally-quantified implication, and there are MUCH worse cases.

How do you feel about verifying Claim 2.6 for all six values in E, which are true/false?[19]

Do you feel uncomfortable saying that the implications with false antecedents are true? Implications are strange, especially when we consider them to involve causality (which we don't in logic). Consider:

CLAIM 2.8: If it rains in Toronto on June 2, 3007, then there are no clouds.

Is Claim 2.8 true or false? Would your answer change if you could wait the required number of decades? What if you waited and June 2, 3007 were a completely dry day in Toronto, is Claim 2.8 true or false?[20]

2.7 Vacuous truth

We use the fact that the empty set is a subset of any set. Let $x \in \mathbb{R}$ (the domain is the real numbers). Is the following implication true or false?

Claim 2.9: If $x^2 - 2x + 2 = 0$, then $x > x + 5$.

A natural tendency is to process $x > x+5$ and think "that's impossible, so the implication is false." However, there is no real number x such that $x^2 - 2x + 2 = 0$, so the antecedent is false for every real x. Whenever the antecedent is false and the consequent is either true or false, the implication as a whole is TRUE. Another way of thinking of this is that the set where the antecedent is true is empty (vacuous), and hence a subset of every set. Such an implication is sometimes called VACUOUSLY TRUE.

In general, if there are no Ps, we consider $P \Rightarrow Q$ to be true, regardless of whether there are any Qs. Another way of thinking of this is that the empty set contains no counterexamples. Use this sort of thinking to evaluate the following claims:[21]

Claim 2.10: All employees making over 80,000 are female.

Claim 2.11: All employees making over 80,000 are male.

Claim 2.12: All employees making over 80,000 have supernatural powers and pink toenails.

2.8 Equivalence

Suppose Al quits the domain E. Consider the claim

Claim 2.13: Every male employee makes between 25,000 and 45,000.

Is Claim 2.13 true? What is its converse?[22] Is the converse true? Draw a Venn diagram. The two properties describe the same set of employees; they are EQUIVALENT. In everyday language, we might say "An employee is male if and only if the employee makes between 25,000 and 45,000." This can be decomposed into two statements:

Claim 2.14: An employee is male if the employee makes between 25,000 and 45,000.

Claim 2.15: An employee is male only if the employee makes between 25,000 and 45,000.

Here are some other everyday ways of expressing equivalence:

- P iff Q ("iff" being an abbreviation for "if and only if").
- P is necessary and sufficient for Q.
- $P \Rightarrow Q$, and conversely.

You may also hear

- P [exactly / precisely] when Q

For example, if our domain is $\mathbb{R}$, you might say "$x^2 + 4x + 4 = 0$ precisely when $x = -2$." Equivalence is getting at the "sameness" (so far as our domain goes) of P and Q. We may define properties P and Q differently, but the same members of the domain have these properties (they define the same sets). Symbolically we write $P \Leftrightarrow Q$. So now

An employee is male $\Leftrightarrow$ he makes between 25,000 and 45,000.

Oddly, our (false) Claim 2.9 is an equivalence, since the implications are vacuously true in both directions: $x^2 - 2x + 2 = 0 \Leftrightarrow x > x + 5$.

2.9 RESTRICTING DOMAINS

Implication, quantification, conjunction ("and," represented by the symbol $\wedge$), and set intersection are techniques that can be used to restrict domains:

- "Every D that is also a P is also a Q" becomes $\forall x \in D, P(x) \Rightarrow Q(x)$, which we use more commonly than the equivalent $\forall x \in D \cap P, Q(x)$
  (What's the difference between this and $\forall x \in D, P(x) \wedge Q(x)$?)
- "Some D that is also a P is also a Q" becomes $\exists x \in D, P(x) \wedge Q(x)$, which we use more commonly than the equivalent $\exists x \in D \cap P, Q(x)$
  (What's the difference between this and $\exists x \in D, P(x) \Rightarrow Q(x)$?)

2.10 CONJUNCTION (AND)

We use $\wedge$ ("and") to combine two sentences into a new sentence that claims that both of the original sentences are true. In our employee database:

CLAIM 2.16: The employee makes less than 75,000 and more than 25,000.

Claim2.16 is true for Al (who makes 60,000), but false for Betty (who makes 500). If we identify the sentences with predicates that test whether objects are members of sets, then the new $\wedge$ predicate tests whether somebody is in both the set of employees who makes less than 75,000 and the set of employees who make more than 25,000 — in other words, in the intersection. Is it a coincidence that $\wedge$ resembles $\cap$ (only more pointy)?

Notice that, symbolically, $P \wedge Q$ is true exactly when both P and Q are true, and false if only one of them is true and the other is false, or if both are false.

We need to be careful with everyday language where the conjunction "and" is used not only to join sentences, but also to "smear" a subject over a compound predicate. In the following sentence the subject "There" is smeared over "pen" and "telephone:"

CLAIM 2.17: There is a pen and a telephone.

If we let O be the set of objects, $p(x)$ mean x is a pen, and $t(x)$ mean x is a telephone, then the obvious meaning of Claim 2.17 is:[23] "There is a pen and there is a telephone." But a pedant who has been observing the trend where phones become increasingly smaller and difficult to use might think Claim 2.17 means:[24] "There is a pen-phone."

Here's another example whose ambiguity is all the more striking since it appears in a context (mathematics) where one would expect ambiguity to be sharply restricted.

> The solutions are:
>
> $x < 10$ and $x > 20$
>
> $x > 10$ and $x < 20$

The author means the union of two sets in the first case, and the intersection in the second. We use $\wedge$ in the second case, and disjunction $\vee$ ("or") in the first case.

2.11 DISJUNCTION (OR)

The disjunction "or" (written symbolically as $\vee$) joins two sentences into one that claims that at least one of the sentences is true. For example,

> The employee is female or makes less than 45,000.

This sentence is true for Flo (she makes 20,000 and is female) and true for Carlos (who makes less than 45,000), but false for Al (he's neither female, nor does he make less than 45,000). If we viewed this "or'ed" sentence as a predicate testing whether somebody belonged to at least one of "the set of employees who are female" or "the set of employees who earn less than 45,000," then it corresponds to the union. As a mnemonic, the symbols $\vee$ and $\cup$ resemble each other. Historically, the symbol $\vee$ comes from the Latin word "vel" meaning or.

We use $\vee$ to include the case where more than one of the properties is true; that is, we use an INCLUSIVE-OR. In everyday English we sometimes say "and/or" to specify the same thing that this course uses "or" for, since the meaning of "or" can vary in English. The sentence "Either we play the game my way, or I'm taking my ball and going home now," doesn't include both possibilities and is an exclusive-or: "one or the other, but not both." An exclusive-or is sometimes added to logical systems (say, inside a computer), but we can use negation and equivalence to express the same thing[25] and avoid the complication of having two different types of "or."

2.12 Negation

We've mentioned negation a few times already, and it is a simple concept, but it's worth examining it in detail. The negation of a sentence simply inverts its truth value. The negation of a sentence P is written as $\neg P$, and has the value true if P was false, and has the value false if P was true.

Negation gives us a powerful way to check our determination of whether a statement is true. For example, we can check that

Claim 2.18: All employees making over 80,000 are female.

is true by verifying that its negation is false. The negation of Claim 2.18 is

Claim 2.19: Not all employees making over 80,000 are female.

We cannot find any employees making over 80,000 that are not female (in fact, we cannot find any employees making over 80,000 at all!), so this sentence must be false, meaning the original must be true.
You should feel comfortable reasoning about why the following are equivalent:

- $\neg(\exists x \in D, P(x) \wedge Q(x)) \Leftrightarrow \forall x \in D, (P(x) \Rightarrow \neg Q(x))$.
  In words, "No P is a Q" is equivalent to "Every P is a non-Q."
- $\neg(\forall x \in D, P(x) \Rightarrow Q(x)) \Leftrightarrow \exists x \in D, (P(x) \wedge \neg Q(x))$.
  In words, "Not every P is a Q" is equivalent to "There is some P that is a non-Q."

Sometimes things become clearer when negation applies directly to the simplest predicates we are discussing. Consider

Claim 2.20: $\forall x \in D, \exists y \in D, P(x, y)$

What does it mean for Claim 2.20 to be false, *i.e.*, $\neg(\forall x \in D, \exists y \in D, P(x, y))$? It means there is some x for which the remainder of the sentence is false:

Claim 2.21: $\neg(\forall x \in D, \exists y \in D, P(x, y)) \Leftrightarrow \exists x \in D, \neg(\exists y \in D, P(x, y))$

So now what does the negated sub-sentence mean? It means there are no y's for which the remainder of the sentence is true:

Claim 2.22: $\exists x \in D, \neg(\exists y \in D, P(x, y)) \Leftrightarrow \exists x \in D, \forall y \in D, \neg P(x, y)$

There is some x that for every y makes $P(x, y)$ false. As negation ($\neg$) moves from left to right, it flips universal quantification to existential quantification, and vice versa. Try it on the symmetrical counterpart $\exists x \in D, \forall y \in D, P(x, y)$, and consider

$$\neg(\exists x \in D, \forall y \in D, P(x,y)) \Leftrightarrow \forall x \in D, \neg(\forall y \in D, P(x,y))$$

If it's not true that there exists an x such that the remainder of the sentence is true, then for all x the remainder of the sentence is false. Considering the remaining subsentence, if it's not true that for all y the remainder of the subsentence is true, then there is some y for which it is false:

$$\neg(\exists x \in D, \forall y \in D, P(x,y)) \Leftrightarrow \forall x \in D, \exists y \in D, \neg P(x,y)$$

For every x there is some y that makes $P(x,y)$ false.
Try combining this with implication, using the rule we discussed earlier, plus DeMorgan's law:

$$\neg(\exists x \in D, \forall y \in D, (P(x,y) \Rightarrow Q(x,y))) \Leftrightarrow \neg(\exists x \in D, \forall y \in D, (\neg P(x,y) \vee Q(x,y)))$$

2.13 SYMBOLIC GRAMMAR

With connectives such as implication ($\Rightarrow$), conjunction ($\wedge$), and disjunction ($\vee$) added to quantifiers, you can form very complex predicates. If you require these complex predicates to be unambiguous, it helps to impose strict conditions on what expressions are allowed. A syntactically correct sentence is sometimes called a well-formed formula (abbreviated wff). Note that syntactic correctness has nothing to do with whether a sentence is true or false, or whether a sentence is open or closed. The syntax (or grammar rules) for our symbolic language can be summarized as follows:

- Any predicate is a wff.
- If P is a wff, so is $\neg P$.
- If P and Q are wffs, so is $(P \wedge Q)$.
- If P and Q are wffs, so is $(P \vee Q)$.
- If P and Q are wffs, so is $(P \Rightarrow Q)$.
- If P and Q are wffs, so is $(P \Leftrightarrow Q)$.
- If P is a wff (possibly open in variable x) and D is a set, then $(\forall x \in D, P)$ is a wff.
- If P is a wff (possibly open in variable x) and D is a set, then $(\exists x \in D, P)$ is a wff.
- Nothing else is a wff.

These rules are recursive, and tell us how we're allowed to build arbitrarily complex sentences in our symbolic language. The first rule is called the base case and specifies the most basic sentence allowed. The rules following the base case are recursive or inductive rules: they tell us how to create a new legal sentence from smaller legal sentences. The last rule is a closure rule, and says we've covered everything.

In practice, we want to avoid writing expressions with many parentheses, so we use PRECEDENCE to disambiguate expressions that are missing parentheses. In the grammar above, precedence decreases from top to bottom. In other words, in the absence of parentheses, parentheses must be added to sub-expressions near the top *before* those near the bottom.
For example, the expression below:

$$\forall x \in D, P(x) \wedge \neg Q(x) \Rightarrow R(x)$$

must be understood as follows — where we have indicated the order in which parentheses were put in, according to the order of precedence above:

$$\big({}_3\forall x \in D, \big({}_2({}_1P(x) \wedge \neg Q(x))_1 \Rightarrow R(x)\big)_2\big)_3$$

You should be able to convert a more loosely-structured predicate into a wff, or a wff into a more loosely-structured predicate, whenever it's convenient.

2.14 Truth tables

Predicates evaluate to either true or false once they are completely specified (all unknown values are filled in). If you build complex predicates from simpler ones, using connectives, it's important to know how to evaluate the complex predicate based on the evaluation of fully-specified variants of the simpler predicates it is built out of. A powerful technique for determining the possible truth value of a complex predicate is the use of TRUTH TABLES. In a truth table, we write all possible truth values for the predicates (how many rows do you need?[26]), and compute the truth value of the statement under each of these truth assignments. Each of the logical connectives yield the following truth tables.

P	$\neg P$
T	F
F	T

P	Q	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

We often break complex statements into simpler substatements, compute the truth value of the substatements, and combine the truth values back into the more complex statements. For example, we can verify the equivalence

$$(P \Rightarrow (Q \Rightarrow R)) \Leftrightarrow ((P \wedge Q) \Rightarrow R)$$

using the following truth table:

P	Q	R	$Q \Rightarrow R$	$P \Rightarrow (Q \Rightarrow R)$	$P \wedge Q$	$(P \wedge Q) \Rightarrow R$	$(P \Rightarrow (Q \Rightarrow R)) \Leftrightarrow ((P \wedge Q) \Rightarrow R)$
T	T	T	T	T	T	T	T
T	T	F	F	F	T	F	T
T	F	T	T	T	F	T	T
T	F	F	T	T	F	T	T
F	T	T	T	T	F	T	T
F	T	F	F	T	F	T	T
F	F	T	T	T	F	T	T
F	F	F	T	T	F	T	T

Since the rightmost column is always true, our statement is a law of logic, and we can use it when manipulating our symbolic statements.

2.15 Tautology, satisfiability, unsatisfiability

Notice that in the previous section, we didn't specify domains or even meanings for P or Q, nor worry about what values might replace unspecified symbols within P or Q. With truth tables we explored all possible "worlds" (configurations of truth assignments to P and Q). This is known as a TAUTOLOGY: you can't dream up a domain, or a meaning for predicates P and Q that provides a counter-example, since the truth tables are identical.

This is different from, say, $(P \Rightarrow Q) \Leftrightarrow (Q \Rightarrow P)$, which will be true for some choice of domain, predicates P and Q, and values of domain elements, so we say this statement is SATISFIABLE. But in this case, there are also choices of domains and/or predicates in which it is false, so it is not a tautology. Be careful: saying that a statement is satisfiable only tells us that it is possible for it to be true, without saying anything about whether or not it is also possible for it to be false (*i.e.*, whether or not it is also a tautology).

What about something for which no domains, predicates, or values can be chosen to make it true? Such a statement would be UNSATISFIABLE (or a CONTRADICTION).

2.16 LOGICAL "ARITHMETIC"

If we identify $\wedge$ and $\vee$ with set intersection and union (for the sets where the predicates they are connecting are true), it's clear that they are ASSOCIATIVE and COMMUTATIVE, so

$$P \wedge Q \Leftrightarrow Q \wedge P \quad \text{and} \quad P \vee Q \Leftrightarrow Q \vee P$$
$$P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R \quad \text{and} \quad P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$$

Maybe a bit more surprising is that we have DISTRIBUTIVE LAWS for each operation over the other:

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$$

We can also simplify expressions using IDENTITY and IDEMPOTENCY laws:

IDENTITY: $P \wedge (Q \vee \neg Q) \Leftrightarrow P \Leftrightarrow P \vee (Q \wedge \neg Q)$

IDEMPOTENCY: $P \wedge P \Leftrightarrow P \Leftrightarrow P \vee P$

DEMORGAN'S LAWS

These laws can be verified either by a truth table, or by representing the sentences as Venn diagrams and taking the complement.
Sentence $s_1 \wedge s_2$ is false exactly when at least one of s_1 or s_2 is false. Symbolically:

$$\neg(s_1 \wedge s_2) \Leftrightarrow (\neg s_1 \vee \neg s_2)$$

Sentence $s_1 \vee s_2$ is false exactly when both s_1 and s_2 are false. Symbolically:

$$\neg(s_1 \vee s_2) \Leftrightarrow (\neg s_1 \wedge \neg s_2)$$

By using the associativity of $\wedge$ and $\vee$, you can extend this to conjunctions and disjunctions of more than two sentences.

IMPLICATION, BI-IMPLICATION, WITH $\neg$, $\vee$, AND $\wedge$

If we shade a Venn diagram so that the largest possible portion of it is shaded without contradicting the implication $P \Rightarrow Q$, we gain some insight into how to express implication in terms of negation and union. The region that we can choose object x from so that $P(x) \Rightarrow Q(x)$ is $\overline{P} \cup Q$ and this easily translates to $\neg P \vee Q$. This gives us an equivalence:

$$(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$$

Now use DeMorgan's law to negate the implication:

$$\neg(P \Rightarrow Q) \Leftrightarrow \neg(\neg P \vee Q) \Leftrightarrow (\neg\neg P \wedge \neg Q) \Leftrightarrow (P \wedge \neg Q)$$

You can use a Venn diagram or some of the laws introduced earlier to show that bi-implication can be written with $\wedge$, $\vee$, and $\neg$:

$$(P \Leftrightarrow Q) \Leftrightarrow ((P \wedge Q) \vee (\neg P \wedge \neg Q))$$

DeMorgan's law tells us how to negate this:

$$\neg(P \Leftrightarrow Q) \Leftrightarrow \neg((P \wedge Q) \vee (\neg P \wedge \neg Q)) \Leftrightarrow \cdots \Leftrightarrow ((\neg P \wedge Q) \vee (P \wedge \neg Q))$$

Transitivity of universally-quantified implication

Consider $\forall x \in D, ((P(x) \Rightarrow Q(x)) \wedge (Q(x) \Rightarrow R(x)))$ (I have put the parentheses to make it explicit that the implications are considered before the $\wedge$). What does this sentence imply if considered in terms of P, Q, and R, the subsets of D where the corresponding predicates are true?[27] We can also work this out using the logical arithmetic rules we introduced above: write $((P(x) \Rightarrow Q(x)) \wedge (Q(x) \Rightarrow R(x))) \Rightarrow (P(x) \Rightarrow R(x))$ using only $\vee, \wedge$, and $\neg$, and show that it is a tautology (always true). Alternatively, use DeMorgan's law, the distributive laws, and anything else that comes to mind to show that the negation of this sentence is a contradiction. Thus, implication is transitive.

A similar transformation is that $\forall x \in D, (P(x) \Rightarrow (Q(x) \Rightarrow R(x))) \Leftrightarrow \forall x \in D, ((P(x) \wedge Q(x)) \Rightarrow R(x))$. Notice this is stronger than the previous result (an equivalence rather than an implication). This statement can be proven with the help of truth tables.

2.17 Summary of manipulation rules

The following is a summary of the basic laws and rules we use for manipulating formal statements. Try proving each of them using Venn diagrams or truth tables.

identity laws	$P \wedge (Q \vee \neg Q) \iff P$
	$P \vee (Q \wedge \neg Q) \iff P$
idempotency laws	$P \wedge P \iff P$
	$P \vee P \iff P$
commutative laws	$P \wedge Q \iff Q \wedge P$
	$P \vee Q \iff Q \vee P$
	$(P \Leftrightarrow Q) \iff (Q \Leftrightarrow P)$
associative laws	$(P \wedge Q) \wedge R \iff P \wedge (Q \wedge R)$
	$(P \vee Q) \vee R \iff P \vee (Q \vee R)$
distributive laws	$P \wedge (Q \vee R) \iff (P \wedge Q) \vee (P \wedge R)$
	$P \vee (Q \wedge R) \iff (P \vee Q) \wedge (P \vee R)$
contrapositive	$P \Rightarrow Q \iff \neg Q \Rightarrow \neg P$
implication	$P \Rightarrow Q \iff \neg P \vee Q$
equivalence	$(P \Leftrightarrow Q) \iff (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
double negation	$\neg(\neg P) \iff P$
DeMorgan's laws	$\neg(P \wedge Q) \iff \neg P \vee \neg Q$
	$\neg(P \vee Q) \iff \neg P \wedge \neg Q$
implication negation	$\neg(P \Rightarrow Q) \iff P \wedge \neg Q$
equivalence negation	$\neg(P \Leftrightarrow Q) \iff \neg(P \Rightarrow Q) \vee \neg(Q \Rightarrow P)$
quantifier negation	$\neg(\forall x \in D, P(x)) \iff \exists x \in D, \neg P(x)$
	$\neg(\exists x \in D, P(x)) \iff \forall x \in D, \neg P(x)$
quantifier distributive laws	$\forall x \in D, P(x) \wedge Q(x) \iff (\forall x \in D, P(x)) \wedge (\forall x \in D, Q(x))$
(where R does not contain variable x)	$\exists x \in D, P(x) \vee Q(x) \iff (\exists x \in D, P(x)) \vee (\exists x \in D, Q(x))$
	$\forall x \in D, R \wedge Q(x) \iff R \wedge (\forall x \in D, Q(x))$
	$\forall x \in D, R \vee Q(x) \iff R \vee (\forall x \in D, Q(x))$
	$\exists x \in D, R \vee Q(x) \iff R \vee (\exists x \in D, Q(x))$
	$\exists x \in D, R \wedge Q(x) \iff R \wedge (\exists x \in D, Q(x))$
variable renaming	$\forall x \in D, P(x) \iff \forall y \in D, P(y)$
(where y does not appear in $P(x)$)	$\exists x \in D, P(x) \iff \exists y \in D, P(y)$

2.18 MULTIPLE QUANTIFIERS

Many sentences we want to reason about have a mixture of predicates. For example

CLAIM 2.23: Some female employee makes more than 25,000.

We can make a few definitions, so let E be the set of employees, $\mathbb{Z}$ be the integers, $\mathbf{sm}(e,k)$ be e makes a salary of more than k, and $\mathbf{f}(e)$ be e is female. Now I could rewrite:

CLAIM 2.23 (SYMBOLICALLY): $\exists e \in E, \mathbf{f}(e) \wedge \mathbf{sm}(e, 25000)$.

It seems a bit inflexible to combine e making a salary, and an inequality comparing that salary to 25000, particularly since we already have a vocabulary of predicates for comparing numbers. We can refine the above expression so that we let $\mathbf{s}(e,k)$ be e makes salary k. Now I can rewrite again:

CLAIM 2.23 (REWRITTEN): $\exists e \in E, \exists k \in \mathbb{Z},\ \mathbf{f}(e) \wedge \mathbf{s}(e,k) \wedge k > 25000$.

Notice that the following are all equivalent to Claim 2.23:

$$\exists k \in \mathbb{Z},\ \exists e \in E, \mathbf{f}(e) \wedge \mathbf{s}(e,k) \wedge k > 25000$$
$$\exists e \in E,\ \mathbf{f}(e) \wedge (\exists k \in \mathbb{Z},\ \mathbf{s}(e,k) \wedge k > 25000)$$

This is because $\wedge$ is commutative and associative, and the two existential quantifiers commute.

2.19 MIXED QUANTIFIERS

If you mix the order of existential and universal quantifiers, you may change the meaning of a sentence. Consider the table below that shows who respects who:

	A	B	C	D	E	F
A	◇					
B		◇	◇	◇	◇	◇
C		◇	◇	◇	◇	◇
D		◇	◇	◇	◇	
E		◇	◇	◇		
F		◇	◇			

If we want to discuss this table symbolically, we can denote the domain of people by P, and the predicate "x respects y" by $r(x,y)$. Consider the following open sentence:

CLAIM 2.24: $\exists x \in P,\ r(x,y)$ (that is "y is respected by somebody")

If we prepended the universal quantifier $\forall y \in P$ to Claim 2.24, would it be true? As usual, check each element of the domain, column-wise, to see that it is.[28] Symbolically,

CLAIM 2.25: $\forall y \in P, \exists x \in P,\ r(x,y)$

or "Everybody has somebody who respects him/her." You can have different x's depending on the y, so although every column has a diamond in some row, it need not be the same row for each column. What would the predicate be that claims that some row works for each column, that a row is full of diamonds?[29] Now we have to check whether there is someone who respects everyone:

CLAIM 2.26: $\exists x \in P, \forall y \in P,\ r(x,y)$

You will find no such row. The only difference between Claim 2.25 and Claim 2.26 is the order of the quantifiers. The convention we follow is to read quantifiers from left to right. The existential quantifier involves making a choice, and the choice may vary according to the quantifiers we have already parsed. As we move right, we have the opportunity to tailor our choice with an existential quantifier (but we aren't obliged to).
Consider this numerical example:

Claim 2.27: $\forall n \in \mathbb{N}, \exists m_1 \in \mathbb{N}, \exists m_2 \in \mathbb{N}, n = m_1 m_2$.

This says that every natural number has two divisors. What does it mean if you switch the order of the existentially quantified variables with the universally quantified variable? Is it still true? What (if anything) would you need to add to say that every natural number has two distinct divisors?[30]

Chapter 2 Notes

[1]Yes, by verifying the claim for each employee.

[2]But contrast the meaning of "differentiable functions are continuous" (every differentiable function is continuous, no exception) with the meaning of "birds fly" (most birds fly, but there are some exceptions).

[3]Betty makes 5,000, which is well-known to be less than 10,000.

[4]Restrict to females, and each one make less than 55,000.

[5]False. Doug and Carlos are counterexamples.

[6]But it is false for males. Al is a counter-example.

[7]False, check the table.

[8]True, check the table.

[9]Yes, since it includes only elements of itself. Don't confuse subset with proper subset.

[10]Yes, indeed it is a subset of every set. The reason is that it contains no element that could be outside another set.

[11]Claim 2.1 depends on who you mean by "The employee." If you specify Al, Claim 2.1 is false, but if you specify Ellen, Claim 2.1 is true. Claim 2.2 is quantified, so it depends on the entire universe of employees. Claim 2.2 is false because you can find at least 1 counterexample.

[12]An untrue implication in the universe we're considering, due to the counter-example Al.

[13]$\overline{P}$ (the complement of P), and Q.

[14]P (we know it's a subset of Q) and $\overline{Q}$ (the complement of Q, we know it's a subset of $\overline{P}$).

[15]Add Grnflx to $F - L$ (F outside L). Now Grnflx is a counter-example to the claim that every female employee makes less than 55,000.

[16]No. Counter-example Grnflx makes the implication false, and adding other data doesn't change this.

[17]Yes. The only Venn diagram that contradicts Claim 2.3 or its contrapositive is one that has at least one element in F outside of L.

[18]No, because there are elements in $L - F$ (Doug and Carlos).

[19]We need to verify the following claims:

- If Al is male, then Al makes less than 55,000.
- If Betty is male, then Betty makes less than 55,000.
- If Carlos is male, then Carlos makes less than 55,000.
- If Doug is male, then Doug makes less than 55,000.
- If Ellen is male, then Ellen makes less than 55,000.
- If Flo is male, then Flo makes less than 55,000.

[20]True, regardless of the cloud situation. In logic $P \Rightarrow Q$ is false exactly when P is true and Q is false. All other configurations of truth values for P and Q are true (assuming that we can evaluate whether P and Q are true or false).

[21]All these claims are true, although possibly misleading. Any claim about elements of the empty set is true, since there are no counterexamples.

[22]Every employee who makes between 25,000 and 45,000 is male.

[23]$\exists x \in O,\ p(x) \wedge \exists y \in O,\ t(y)$, or even $\exists x \in O,\ \exists y \in O,\ p(x) \wedge t(y)$.

[24]$\exists x \in O,\ p(x) \wedge t(x)$

[25]"P exclusive-or Q" is the same as "P not-equivalent-to Q."

[26]If you have n predicates, you need 2^n rows (every combination of T and F).

[27]It implies that P is a subset of R, since $P \subseteq Q$ and $Q \subseteq R$. It is not equivalent, since you can certainly have $P \subseteq R$ without $P \subseteq Q$ or $R \subseteq Q$.

[28]True, there's a diamond in every column.

[29]If we were thinking of the row corresponding to x, then $\forall y \in P,\ r(x, y)$.

[30]$\forall n \in \mathbb{N}, \exists m_1 \in \mathbb{N}, \exists m_2 \in \mathbb{N}, n = m_1 m_2 \wedge m_1 \neq m_2$. Not true for $n = 1$.

Chapter 3

Proofs

3.1 What is a proof?

A PROOF is an argument that convinces someone who is logical, careful and precise. The form and detail of a proof can depend on the audience (for example, whether our audience has as much general math knowledge, and whether we're writing in English or our symbolic form), but the fundamentals are the same whether we're talking mathematics, computer science, physical sciences, philosophy, or writing an essay in literature class. A proof communicates what (and how) someone understands, to save others time and effort. IF YOU DON'T UNDERSTAND WHY SOMETHING IS TRUE, DON'T EXPECT TO BE ABLE TO PROVE IT!
How do you go about writing a proof? Generally, there are two steps or phases to creating a proof:

1. Understanding why something is true.

   This step typically requires some creativity and multiple attempts until an approach works. You should ask yourself why you are convinced something is true, and try to express your thoughts precisely and logically. This step is the most important (and requires the most effort), and can be done in the shower or as you lie awake in bed (the two most productive thinking spots).

   Sometimes we call this FINDING A PROOF.

2. Writing up your understanding.

   Be careful and precise. Every statement you write should be true in the context it's written. It is often helpful to use our formal symbolic form, to ensure you're careful and precise. Often you will detect errors in your understanding, and it's common to then go back to step 1 to refine your understanding.

   This is when we are WRITING UP A PROOF.

Sometimes these steps can be combined, and often these steps feedback on each other. As we try to write up our understanding, we discover a flaw, return to step 1 and refine our understanding, and try writing again.

Students are often surprised that most of the work coming up with a proof is understanding why something is true. If you go back to our definition of what is a proof, this should be obvious: to convince someone, we first need to convince ourselves and order our thoughts precisely and logically. You will see that once we gain a good understanding, proofs nearly write themselves.

Taxonomy of results

A LEMMA is a small result needed to prove something we really care about. A THEOREM is the main result that we care about (at the moment). A COROLLARY is an easy (or said to be easy) consequence of another result. A CONJECTURE is something suspected to be true, but not yet proven.[1] An AXIOM is something we assert to be true, without justification — usually because it is "self-evident."[2]

3.2 DIRECT PROOF OF UNIVERSALLY-QUANTIFIED IMPLICATION

We want to make convincing arguments that a statement is true. We're allowed (forced, actually) to use previously proven statements and axioms. For example, if D is the set of real numbers, then we have plenty of rules about arithmetic and inequalities in our toolbox. From these statements, we want to extend what we know, eventually to include the statement we're trying to prove. Let's examine how we might go about doing this.
Consider an implication we would like to prove that is of the form:

C1: $\forall x \in D, p(x) \Rightarrow q(x)$

Many already-known-to-be-true statements are universally quantified implications, having an identical structure to C1. We'd like to find among them a chain:

C2.0: $\forall x \in D, p(x) \Rightarrow r_1(x)$

C2.1: $\forall x \in D, r_1(x) \Rightarrow r_2(x)$

$\vdots$

C2.N: $\forall x \in D, r_n(x) \Rightarrow q(x)$

This, in n steps, proves C1, using the transitivity of implication.

A more flexible way to summarize that the chain C2.0,...,C2.N proves C1 is to cite the intermediate implications that justify each intermediate step. Here you write the proof that $\forall x \in D, p(x) \Rightarrow q(x)$ as:

Assume $x \in D$. # x is a generic element of D
  Assume $p(x)$. # x has property p, the antecedent
    Then $r_1(x)$. # by C2.0
    Then $r_2(x)$. # by C2.1
      $\vdots$
    Then $q(x)$. # by C2.N
  Then $p(x) \Rightarrow q(x)$. # assuming antecedent leads to consequent
Then $\forall x \in D, p(x) \Rightarrow q(x)$. # we only assumed x is a generic D

This form emphasizes what each existing result adds to our understanding. And when it's obvious which result was used, we can just avoid mentioning it (but be careful, one person's obvious is another's mystery).

Although this form seems to talk about just one particular x, by not assuming anything more than $x \in D$ and $p(x)$, it applies to every $x \in D$ with $p(x)$.

The indentation shows the scope of our assumptions. When we assume that $x \in D$, we are in the "world" where x is a generic element of D. Where we assume $p(x)$, we are in the "world" where $p(x)$ is assumed true, and we can use that to derive consequences.

HUNTING THE ELUSIVE DIRECT PROOF

In general, the difficulty with direct proof is there are lots of known results to consider. The fact that a result is true may not help your particular line of argument (there are many, many, many true but irrelevant facts). In practice, to find a chain from $p(x)$ to $q(x)$, you gather two lists of results about x:

1. results that $p(x)$ implies, and
2. results that imply $q(x)$

Your fervent hope is that some result appears on both lists, since then you'll have a chain.

$p(x)$
$r_1(x)$
$r_2(x)$
$\vdots$
$s_2(x)$
$s_1(x)$
$q(x)$

Anything that one of the r_i implies can be added to the first list. Anything that implies one of the s_i can be added to the second list.

What does this look like in pictures? In Venn diagrams we can think of the r_i as sets that contain p and may, or may not, be contained in q (the ones that aren't contained in q are dead ends). On the other hand, the s_i are contained in q and may, or may not, contain p (the ones that don't are dead ends). We hope to find a patch of containment from p to q. Another way to visualize this is by having the r_i represented as a tree. In one tree we have root p, with children being the r_i that p implies, and their children being results they imply. In a second tree we have root q, with children being the results that imply q, and their children being results that imply them. If the two trees have a common node, we have a chain.

Are you done when you find a chain? No, you write it up, tidying as you go. Remove the results that don't contribute to the final chain, and cite the results that take you to each intermediate link in the chain.

What do $\wedge$ and $\vee$ do?

Now your two lists have the form

$\forall x \in D, p(x) \Rightarrow (r_1(x) \wedge r_2(x) \wedge \cdots \wedge r_m(x))$
$\forall x \in D, (s_k(x) \vee \cdots \vee s_1(x)) \Rightarrow q(x)$

Since $p(x)$ implies any "and" of the r_i, you can just collect them in your head until you find a known result, say $r_1(x) \wedge r_2(x) \Rightarrow r_k(x)$, and then add $r_k(x)$ to the list. On the other hand, if you have a result on the first list of the form $r_1(x) \wedge r_2(x)$, you can add them separately to the list. On the second list, use the same approach but substitute $\vee$ for $\wedge$. Any result on the first list can be spuriously "or'ed" with anything: $r_1(x) \Rightarrow (r_1(x) \vee l(x))$ is always true. On the second list, we can spuriously "and" anything, since $(s_1(x) \wedge l(x)) \Rightarrow s_1(x)$.

If we have a disjunction $r_1(x) \vee r_2(x)$ on the first list, we can use it if we have a result that $(r_1(x) \vee r_2(x)) \Rightarrow q(x)$, or the pair of results $r_1(x) \Rightarrow q(x)$, and $r_2(x) \Rightarrow q(x)$.

3.3 An odd example of direct proof

Suppose you are asked to prove that every odd natural number has a square that is odd. Typically we don't see all the links in the chain from "n is odd" to "n^2 is odd" instantly, so we engage in thoughtful wishing (like wishful thinking, only with a much better reputation). We start by writing the outline of the proof we would like to have, to clarify what information we've got, what we lack, and hope to fill in the gaps:

Assume $n \in \mathbb{N}$.
  Assume n is odd.
    $\vdots$
    Then n^2 is odd.
  Then n is odd $\Rightarrow n^2$ is odd.
Then $\forall n \in \mathbb{N}, n$ is odd $\Rightarrow n^2$ is odd.

Start scratching away at both ends of the $\vdots$ (the bit that represents the chain of results we need to fill in). What does it mean for n^2 to be odd? Well, if there is a natural number k such that $n^2 = 2k + 1$, then n^2 is odd (by definition of odd numbers). Add that to the end of the list. Similarly, if n is odd, then there is a natural number j such that $n = 2j + 1$ (by definition of odd numbers). It seem unpromising to take the square root of $2k+1$, so instead carry out the almost-automatic squaring of $2j + 1$. Now, on our first list, we have that, for some natural number j, $n^2 = 4j^2+4j+1$. Using some algebra (distributivity of multiplication over addition), this means that for some natural number j, $n^2 = 2(2j^2 + 2j) + 1$. If we let k from our second list be $2j^2 + 2j$, then we certainly satisfy the restriction that k be a natural number (they are closed under multiplication and addition), and we have linked the first list to the second. Here's an example of how to format your finished chain:

Assume $n \in \mathbb{N}$. # n is a generic natural number
  Assume n is odd. # n a typical odd natural number
    Then, $\exists j' \in \mathbb{N}$, $n = 2j' + 1$. # by definition of n odd
    Let $j \in \mathbb{N}$ be such that $n = 2j + 1$. # name it j
    Then $n^2 = 4j^2 + 4j + 1 = 2(2j^2 + 2j) + 1$. # definition of n^2 and some algebra
    Then $\exists k \in \mathbb{N}, n^2 = 2k + 1$. # $2j^2 + 2j \in \mathbb{N}$, since $\mathbb{N}$ closed under $+, \times$
    Then n^2 is odd. # by definition of n^2 odd
  Then n is odd $\Rightarrow n^2$ is odd. # when I assumed n odd, I derived n^2 odd
Then $\forall n \in \mathbb{N}$, n odd $\Rightarrow$ n^2 odd. # since n was a generic natural number

How about the converse, $\forall n \in \mathbb{N}$, if n^2 is odd, then n is odd. If we try creating a chain, it seems a bit as though the natural direction is wrong: somehow we'd like to go from q back to p. What equivalent of an implication allows us to do this?[3] You set up the proof of the contrapositive of the converse (whew!) very similarly to the proof above, mostly changing "odd" to "even." Try it out.

ANOTHER EXAMPLE OF DIRECT PROOF

Let $\mathbb{R}$ be the set of real numbers. Prove:

$\forall x \in \mathbb{R}, x > 0 \Rightarrow 1/(x + 2) < 3$

Structure the proof as before:

Assume $x \in \mathbb{R}$. # x is a typical real number
  Assume $x > 0$. # antecedent
    $\vdots$ # prove $1/(x + 2) < 3$
    Then $1/(x + 2) < 3$. # get here somehow
  Then $x > 0 \Rightarrow 1/(x + 2) < 3$. # assume antecedent, derived consequent
Then $\forall x \in \mathbb{R}, x > 0 \Rightarrow 1/(x + 2) < 3$. # only assume x was a typical element of $\mathbb{R}$

Of course, you need to unwrap the sub-proof that $1/(x + 2) < 3$:

Assume $x \in \mathbb{R}$. # x is a typical element of $\mathbb{R}$
  Assume $x > 0$. # antecedent
    Then $x + 2 > 2$. # $x > 0$, add 2 to both sides
    Then $1/(x + 2) < 1/2$. # reciprocals reverse inequality, and are defined for numbers > 2
    Then $1/(x + 2) < 3$. # since $1/(x + 2) < 1/2$ and $1/2 < 3$
  Then $x > 0 \Rightarrow 1/(x + 2) < 3$. # assumed antecedent, derived consequent
Then $\forall x \in \mathbb{R}, x > 0 \Rightarrow 1/(x + 2) < 3$. # x was assumed to be a typical element of $\mathbb{R}$

Is the converse true (what is the converse)?[4]

3.4 INDIRECT PROOF OF UNIVERSALLY-QUANTIFIED IMPLICATION

Recall that $p \Rightarrow q$ is equivalent to its contrapositive, $\neg q \Rightarrow \neg p$. This means that proving one proves the other. This is called an "indirect proof." The outline format of an indirect proof of $\forall x \in D, p(x) \Rightarrow q(x)$ is

Assume $x \in D$. # x is a typical element of D
  Assume $\neg q(x)$. # negation of the CONSEQUENT!
    $\vdots$
    Then $\neg p(x)$. # negation of the ANTECEDENT!
  Then $\neg q(x) \Rightarrow \neg p(x)$. # assuming $\neg q(x)$ leads to $\neg p(x)$
  Then $p(x) \Rightarrow q(x)$. # implication is equivalent to contrapositive
Then $\forall x \in D, p(x) \Rightarrow q(x)$. # x was a typical element of D

This is a useful approach, for example, in proving that $\forall n \in \mathbb{N}, n^2$ is odd $\Rightarrow n$ is odd.

3.5 DIRECT PROOF OF UNIVERSALLY-QUANTIFIED PREDICATE

When no implication is stated, then we don't assume (suppose) anything about x other than membership in the domain. For example, $\forall x \in D, p(x)$ has this proof structure:

Assume $x \in D$.
  $\vdots$ # prove $p(x)$
  Then $p(x)$.
Then $\forall x \in D, p(x)$. # x was assumed to be a typical element of D

3.6 PROOF BY CONTRADICTION

Sometimes you want to prove a conclusion, Q, without any suitable hypothesis, P to imply it. One approach is to say "if everything we already know is true is assumed, then Q follows." How do you choose which particular portion of "everything we already know is true" to focus on? Let logic help focus your argument.

Symbolically you can represent "everything we already know is true" as a huge conjunction of statements, $P = P_1 \wedge P_2 \wedge \cdots \wedge P_m$. So now we aim to prove $P \Rightarrow Q$ using the CONTRAPOSITIVE: $\neg Q \Rightarrow \neg P$. Start by assuming that Q is FALSE, and then show that something you already know to be true must be false — a contradiction! Since $P = P_1 \wedge P_2 \wedge \cdots \wedge P_m$ is a huge conjunction of statements, its negation is a huge disjunction $\neg P = \neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_m$, so you don't need to know in advance which of them is contradicted. You just follow your (educated) nose. Here's the general format:

Assume $\neg Q$. # in order to derive a contradiction
  $\vdots$ # some steps leading to a contradiction, say $\neg P_j$
  Then $\neg P$. # contradiction, since P is known to be true
Then Q. # since assuming $\neg Q$ leads to contradiction

Euclid used this technique over 2,000 years ago to prove that there are infinitely many prime numbers. Before looking at Euclid's proof, you might experiment with proving this fact directly.
Let's start by naming some of the sets/predicates we'll need for this proof:

- $P = \{p \in \mathbb{N} : p \text{ has exactly two factors}\}$
- SP: $\forall n \in \mathbb{N}, |P| > n$

In spite of appearances, SP is not a good candidate for mathematical induction (which we'll see later in this course). However, let's try $\neg$SP:

Assume $\neg$SP: $\exists n \in \mathbb{N}, |P| \leqslant n$. # to derive a contradiction
  Then there is a finite list, $p_1, \ldots, p_k$ of elements of P. # at most n elements in the list
  Then I can take the product $p' = p_1 \times \cdots \times p_k$. # finite products are well-defined
  Then p' is the product of some natural numbers 2 and greater. # 0, 1 aren't primes, 2, 3 are
  Then $p' > 1$. # p' is at least 6
  Then $p' + 1 > 2$. # add 1 to both sides
  Then $\exists p \in P, p$ divides $p' + 1$. # every integer > 2 (such as $p' + 1$) has a prime divisor
  Let $p_0 \in P$ be such that p_0 divides $p' + 1$. # instantiate existential
  Then p_0 is one of $p_1, \ldots, p_k$. # by assumption, the only primes
  Then p_0 divides $p' + 1 - p' = 1$. # a divisor of each term divides difference
  Then $1 \in P$. Contradiction! # 1 is not prime
Then SP. # "assume $\neg$SP" leads to a contradiction

3.7 Direct proof structure of the existential

Consider the example $\exists x \in \mathbb{R}, x^3 + 2x^2 + 3x + 4 = 2$. Since this is the existential, we need only find a single example to show that the statement is true. We structure the proof as follows:

Let $x = -1$. # choose a particular element that will work
Then $x \in \mathbb{R}$. # verify that the element is in the domain
Then $x^3 + 2x^2 + 3x + 4 = (-1)^3 + 2(-1)^2 + 3(-1) + 4 = -1 + 2 - 3 + 4 = 2$. # substitute -1 for x
Then $\exists x \in \mathbb{R}$, $x^3 + 2x^2 + 3x + 4 = 2$. # we gave an example

The general form for a direct proof of $\exists x \in D, p(x)$ is:

Let $x = \ldots$ # choose a particular element of the domain
Then $x \in D$. # this may be obvious, otherwise prove it
  $\vdots$ # prove $p(x)$
Then $p(x)$. # you've shown that x satisfies p
$\exists x \in D, p(x)$. # introduce existential

3.8 Multiple quantifiers, implications, and conjunctions

Consider $\forall x \in D, \exists y \in D, p(x, y)$. The corresponding proof structure is:

Assume $x \in D$. # typical element of D
  Let $y_x = \ldots$ # choose an element that works
    $\vdots$
  Then $y_x \in D$. # verify that $y \in D$
    $\vdots$
  Then $p(x, y_x)$. # y satisfies $p(x, y)$
  Then $\exists y, p(x, y)$. # introduce existential
Then $\forall x \in D, \exists y \in D, p(x, y)$. # introduce universal

Here's a concrete example. Suppose we have a mystery function f, mystery constants a and l, and the following statement (I have added parentheses to indicate the conventional parsing):

$$\forall e \in \mathbb{R},\, e > 0 \Rightarrow (\exists d \in \mathbb{R}, d > 0 \wedge (\forall x \in \mathbb{R},\, 0 < |x - a| < d \Rightarrow |f(x) - l| < e))$$

If we want to prove this TRUE, structure the proof as follows...[5]

If we want to prove the statement FALSE, we first negate it, and then use one of our proof formats (I use the equivalences $\neg(p \Rightarrow q) \Leftrightarrow (p \wedge \neg q)$ and $\neg(p \wedge q) \Leftrightarrow (p \Rightarrow \neg q)$):

$$\exists e \in \mathbb{R},\, e > 0 \wedge \forall d \in \mathbb{R},\, d > 0 \Rightarrow \exists x \in \mathbb{R},\, 0 < |x - a| < d \wedge |f(x) - l| \geqslant e$$

Of course, this negation involved several applications of rules we already know, and now its proof may be written step-by-step. Notice that, in the middle of our proof, we had a "$\wedge$" to prove.

3.9 EXAMPLE OF PROVING A STATEMENT ABOUT A SEQUENCE

Consider the statement:

CLAIM 3.1: $\exists i \in \mathbb{N}, \forall j \in \mathbb{N}, a_j \leqslant i \Rightarrow j < i$

and the sequence:

(A1) $0, 1, 4, 9, 16, 25, \ldots$

We'll use the convention that sequences are indexed by natural numbers (recall that $\mathbb{N} = \{0, 1, 2, \ldots\}$, starting at zero just as computers count) and a_i is the element of the sequence indexed by i, so $a_0 = 0$, $a_1 = 1$, $a_2 = 4$. Looking at the pattern of (A1), we can write a "closed form" formula for a_i.[6]

We should of course try to understand 3.1, by putting it in natural English, picturing tables and diagrams, thinking of code that could check it, trying it on various examples, *etc.* To understand whether it is true or false for (A1) we should use this understanding, including tracing it. But let's focus on the form that a proof that 3.1 is true could take. This may even help us understand 3.1.
We have been justifying existentials with an example. So, our proof should start off something like:

Let $i =$ ____. Then $i \in \mathbb{N}$.
⋮

We leave ourselves a blank to fill in: a specific value of i. We also need to make sure the i is in $\mathbb{N}$. Often it will be obvious and we will simply note it. If not, we'll actually need to put in a proof.

Next, we need to prove something for all j in $\mathbb{N}$. As a syntactic convenience, we prove something for all j's in $\mathbb{N}$ by proving it for some *unknown* j in $\mathbb{N}$. If we're careful to not assume anything about which j we have, our proof will handle all j's.

Let $i =$ __. Then $i \in \mathbb{N}$. # choose a helpful one
Assume $j \in \mathbb{N}$. # j is a typical element of $\mathbb{N}$
⋮

Notice this time we *assume* j is in $\mathbb{N}$. You can imagine $\exists$ and $\forall$ as part of a game:

- $\exists x \in D$: We get to pick x, but have to follow the rules and pick from D.
- $\forall x \in D$: Our opponent will pick x, but we can assume they will follow the rules and pick from D. We can't make any assumptions here about which one from D they will pick.

Going back to our proof structure, we have:

Let $i =$ __. Then $i \in \mathbb{N}$.
Assume $j \in \mathbb{N}$. # typical element of $\mathbb{N}$
  Assume $a_j \leqslant i$.
    ⋮
  Then $j < i$.

We leave ourselves room (the $\vdots$) for a proof of $j < i$. Once we fill in a value of i, the proof of $j < i$ may use three facts: the value we chose for i, $j \in \mathbb{N}$, and $a_j \leqslant i$.

After a little thought, we decide that setting $i = 2$ is a good idea, since then $a_j \leqslant i$ is only true for $j = 0$ and $j = 1$, and these are smaller than 2. A bit of experimentation shows that the contrapositive, $\neg(j < i) \Rightarrow \neg(a_j \leqslant a_i)$ is a bit easier to work with.

Let $i = 2$. Then $i \in \mathbb{N}$. # $2 \in \mathbb{N}$
Assume $j \in \mathbb{N}$. # typical element of $\mathbb{N}$
  Assume $\neg(j < i)$. # antecedent for contrapositive
    Then $j \geqslant 2$. # negation of $j < i$ when $i = 2$
    Then $a_j = j^2 \geqslant 2^2 = 4$. # since $a_j = j^2$, and $j \geqslant 2$
    Then $a_j > 2$. # since $4 > 2$
  Then $\neg(j < i) \Rightarrow \neg(a_j \leqslant 2)$. # assuming antecedent leads to consequent
  Then $a_j \leqslant 2 \Rightarrow j < i$. # implication equivalent to contrapositive
Then $\forall j \in \mathbb{N}, a_j \leqslant i \Rightarrow j < i$. # introduce universal
Then $\exists i \in \mathbb{N}, \forall j \in \mathbb{N}, a_j \leqslant i \Rightarrow j < i$. # introduce existential

3.10 EXAMPLE OF DISPROVING A STATEMENT ABOUT A SEQUENCE

Consider now the statement:

CLAIM 3.2: $\exists i \in \mathbb{N}, \forall j \in \mathbb{N}, j > i \Rightarrow a_j = a_i$

and the sequence:

(A2) $0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, \ldots$

Let's disprove it. Is disproof a whole new topic? Thankfully no. We simply prove the negation:

CLAIM 3.2′: $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j > i \wedge a_j \neq a_i$

As usual, we sketch in the outline of the proof first:

Assume $i \in \mathbb{N}$.
  Let $j =$ ____. Then $j \in \mathbb{N}$.
    $\vdots$
  Then $j > i \wedge a_j \neq a_i$.
  Then $\exists j \in \mathbb{N}, j > i \wedge a_j \neq a_i$. # introduction of existential
Then $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j > i \wedge a_j \neq a_i$. # introduction of universal

Our opponent picks i, but we get to pick j. And we are allowed to make j depend on i. Unfortunately, while writing up the proof we can't wait for someone to actually pick i. So how does it help us? We get to describe a general strategy for how we would pick a particular j if we knew which particular i. In other words, j can be described as function of i.

In programming terms, i is in scope when we pick j: it has been declared and can be seen from where we declare j. Notice that j is not in scope when we declare i: so when we picked i for 3.1, we weren't allowed to use j. If we write a program that uses a variable before it's declared and initialized, the program doesn't even compile. This is a major error. If you write a proof that does this, it will almost certainly be wrong — and you will most likely lose a lot of marks!

Now we are left with proving $j > i \wedge a_j \neq a_i$ (notice we wrote this at the bottom... we must have been thinking ahead). What form does the proof of a conjunction take?[7]

Assume $i \in \mathbb{N}$.
  Let $j =$ ____. Then $j \in \mathbb{N}$.
    ⋮
  Then $j > i$.
    ⋮
  Then $a_j \neq a_i$.
  Then $j > i \wedge a_j \neq a_i$.
  Then $\exists j \in \mathbb{N}, j > i \wedge a_j \neq a_i$.
Then $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j > i \wedge a_j \neq a_i$.

To finish this off, we need to choose a value for j. If we choose wisely, the rest of the proof falls into place.[8] What elementary property of arithmetic will we require?[9]

3.11 Non-boolean function example

Non-boolean functions cannot take the place of predicates (since predicates are expected to return a true or false value) in a proof. How should non-boolean functions be used? Define $\lfloor x \rfloor : \mathbb{R} \to \mathbb{R}$ by:

> $\lfloor x \rfloor$ ("floor of x") is the largest integer $\leqslant x$.

Now we can form the statement:

Claim 3.3: $\forall x \in \mathbb{R}, \lfloor x \rfloor < x + 1$

It makes sense to apply $\lfloor x \rfloor$ to elements of our domain, or variables that we have introduced, and to evaluate it in predicates such as "$<$" but $\lfloor x \rfloor$ itself is not a variable, nor a sentence, nor a predicate. We can't (sensibly) say things such as $\forall \lfloor x \rfloor \in \mathbb{R}$ or $\forall x \in \mathbb{R}, \lfloor x \rfloor \vee \lfloor x + 1 \rfloor$. The structure of 3.3 is a direct proof of a universally-quantified predicate:[10]

Assume $x \in \mathbb{R}$. # x is a typical element of $\mathbb{R}$
  Then $\lfloor x \rfloor$ is the largest integer $\leqslant x$, so $\lfloor x \rfloor \leqslant x$. # definition of floor
  Since $x < x + 1$, $\lfloor x \rfloor < x + 1$. # transitivity of $<$
Then $\forall x \in \mathbb{R}, \lfloor x \rfloor < x + 1$. # since x was a typical element of $\mathbb{R}$

In some cases you need to break down a statement such as "$\lfloor x \rfloor$ is the largest integer $\leqslant x$":

$$\lfloor x \rfloor \in \mathbb{Z} \wedge \lfloor x \rfloor \leqslant x \wedge (\forall z \in \mathbb{Z}, z \leqslant x \Rightarrow z \leqslant \lfloor x \rfloor)$$

We didn't need all three parts of the definition for our proof above, and in practice we don't always have to return to definitions when dealing with functions. For example, we may have an existing result, such as:

$$\forall x \in \mathbb{R}, \lfloor x \rfloor > x - 1$$

How would you prove this result, using the three-part version of the definition of $\lfloor x \rfloor$?

3.12 Substituting known results

Every proof would become unmanageably long if we had to include "inline" all the results that it depended on. We inevitably refer to standard results that are either universally known (among math wonks) or can easily be looked up. Sometimes we need to prove a small technical result in order to prove something larger. You may view the smaller result as a helper method (usually returning boolean results) that you use to build a larger method (your bigger proof). To make things modular, you should be able to "call" or refer to the smaller result. An example occurs if we want to re-cycle something proved earlier:

THEOREM 1: $\forall x \in \mathbb{R}, x > 0 \Rightarrow 1/(x+2) < 3$.

We want to use this in proving $\forall y \in \mathbb{R}, y \neq 0 \Rightarrow 1/(y^2+2) < 3$. The template to fill in is

Assume $y \in \mathbb{R}$.
  Assume $y \neq 0$.
    $\vdots$
    Then $1/(y^2+2) < 3$.
  Then $y \neq 0 \Rightarrow 1/(y^2+2) < 3$.
Then $\forall y \in \mathbb{R}, y \neq 0 \Rightarrow 1/(y^2+2) < 3$.

Now we have to fill in the $\vdots$ part:

Assume $y \in \mathbb{R}$. # y is a typical element of $\mathbb{R}$
  Assume $y \neq 0$. # y positive
    Then $y^2 \in \mathbb{R}$ and $y^2 \geqslant 0$. # $\mathbb{R}$ closed under $\times$, squares of non-zero reals
    Then $y^2 > 0$, since $y^2 \neq 0$ and $y^2 \geqslant 0$. # only real number whose square is 0 is 0
    Then $1/(y^2+2) < 3$. # by Theorem 1
  Then $y \neq 0 \Rightarrow 1/(y^2+2) < 3$. # introduction of $\Rightarrow$
Then $\forall y \in \mathbb{R}, y \neq 0 \Rightarrow 1/(y^2+2) < 3$. # introduction of universal

3.13 PROOF BY CASES

To prove $A \Rightarrow B$, it can help to treat some A's differently than others. For example, to prove that $x^2 + x$ is even for all integers x, you might proceed by noting that $x^2 + x$ is equivalent to $x(x+1)$. At this point our reasoning has to branch: at least one of the factors x or $x+1$ is even (for integer x), but we can't assume that a particular factor is even for every integer x. So we use proof by cases.[11]

This is a special case of an "or" clause being the antecedent of an implication, *i.e.*, if you want to prove $(A_1 \vee A_2 \vee \cdots \vee A_n) \Rightarrow B$. This could happen if, along the way to proving $A \Rightarrow B$ you use the fact that $A \Rightarrow (A_1 \vee \cdots \vee A_n)$. Now you need to prove $A_1 \Rightarrow B$, $A_2 \Rightarrow B, \ldots, A_n \Rightarrow B$. Notice that in setting this up it is not necessary that the A_i be disjoint (mutually exclusive), just that they cover A (think of A being a subset of the union of the A_i). One way to generate the cases is to break up the domain $D = D_1 \cup \cdots \cup D_n$, so A_i is the predicate that corresponds to the set $D_i \cap A$. Now you have an equivalence, $A \Leftrightarrow A_1 \vee \cdots \vee A_n$. A very common case occurs when the domain partitions into two parts, $D = D_1 \cup \overline{D_1}$, so you can rewrite A as $(A \wedge D_1) \vee (A \wedge \neg D_1)$ — we're abusing the notation slightly here by treating D_1 both as a set and as a predicate, as we've done before.
Here's the general form of proving something by cases:

$A \vee B$
Case 1: Assume A.
  $\vdots$
  Then C.
Case 2: Assume B.
  $\vdots$
  Then C.
Since $A \vee B$ and in both (all) cases we concluded C, then C.

Remember that we need one case for each disjunct, so if we knew $A_1 \vee \cdots \vee A_n$, we'd need n cases.

When you're reading (or writing) proofs, often the word "assume" is omitted when defining the case. Though it might say "Case $x < k$," remember that $x < k$ is an assumption, thus opens a new indentation (scope) level.

Law of the excluded middle

Often we want to proceed by cases, but don't have a disjunction handy to use. We can always introduce one using the Law of the Excluded Middle. This law of logic states that a formula is either TRUE or FALSE—there's nothing between (or "in the middle"). Thus, for any formula P, the following is sure to be true:

$$P \vee \neg P$$

In your proof, you can then split into two cases depending on whether P is true or false. Just be sure to negate P correctly!

Example proof using cases

Suppose we wanted to prove the following statement: if n is an integer then $n^2 + n$ is even.
Let's formalize what we mean by the term "integer n is even":

For $n \in \mathbb{Z}$, let even(n) mean $\exists k \in \mathbb{Z}, n = 2k$.

Let's formalize what we're proving:

Claim 3.4: $\forall n \in \mathbb{Z}, \exists k \in \mathbb{Z}, n^2 + n = 2k$.

Noticing that $n^2 + n = n(n+1)$, we consider whether n is odd or even. We know that every integer is either odd or even, so let's state this formally:

(*) $\forall n \in \mathbb{Z}, (\exists k \in \mathbb{Z}, n = 2k+1) \vee (\exists k \in \mathbb{Z}, n = 2k)$.

Now to the proof of our claim. In it we will know that an existential is true, and we will want to use that knowledge. We may ask the existential to "return" an example element, which we get to name and use (we name it k_0 so that it won't conflict with any other elements we're talking about).

Assume $n \in \mathbb{Z}$. # n is a typical natural number
  Then $(\exists k \in \mathbb{Z}, n = 2k+1) \vee (\exists k \in \mathbb{Z}, n = 2k)$. # by (*), $n \in \mathbb{Z}$
  Case 1: Assume $\exists k \in \mathbb{Z}, n = 2k+1$.
    Let $k_0 \in \mathbb{Z}$ be such that $n = 2k_0 + 1$. # instantiate existential
    Then $n^2 + n = n(n+1) = (2k_0+1)(2k_0+2) = 2(2k_0+1)(k_0+1)$.
    Then $\exists k \in \mathbb{Z}, n^2 + n = 2k$. # $k = (2k_0+1)(k_0+1) \in \mathbb{Z}$
  Case 2: Assume $\exists k \in \mathbb{Z}, n = 2k$.
    Let $k_0 \in \mathbb{Z}$ be such that $n = 2k_0$. # instantiate existential
    Then $n^2 + n = n(n+1) = 2k_0(2k_0+1) = 2[k_0(2k_0+1)]$.
    Then $\exists k \in \mathbb{Z}, n^2 + n = 2k$. # $k = k_0(2k_0+1) \in \mathbb{Z}$
  Then $\exists k \in \mathbb{Z}, n^2 + n = 2k$. # true in all (both) possible cases
Then $\forall n \in \mathbb{Z}, \exists k \in \mathbb{Z}, n^2 + n = 2k$. # introduction of universal

Proving $\vee$ using cases

Let's prove that the square of an integer is a triple or one more than a triple.

Claim 3.5: $\forall n \in \mathbb{N}, (\exists k \in \mathbb{N}, n^2 = 3k) \vee (\exists k \in \mathbb{N}, n^2 = 3k+1)$.

If we know $P \vee Q$, we can prove a disjunction $R \vee S$ by cases, as follows:

$P \vee Q$
Case 1: Assume P.
  ⋮
  Then R.

Case 2: Assume Q.
$\vdots$
Then S.
Thus $R \vee S$.[12]

If we already have some $P \vee Q$ we can use, then those are the obvious cases to consider, though we still have to decide between the two ways of pairing them up with R and S. In general though, picking P and Q that work depends completely on context. When constructing proof structures, a standard strategy is to use $\neg P$ for Q: the Law of the Excluded Middle ensures this is true, and it is the simplest yet still general structure.

This of course generalizes to more than two cases: if we know $P_1 \vee P_2 \vee \cdots \vee P_n$, and we want to prove $Q_1 \vee \cdots \vee Q_m$, then we can do cases for each P_i, in each case proving a Q_j. We don't have to prove all the Q_j, and we can prove some of them in more than one case.
To prove our claim, we want to use part of the Remainder Theorem:

(*) $\forall n \in \mathbb{N}, (\exists k \in \mathbb{N}, n = 3k \vee n = 3k+1 \vee n = 3k+2)$

We now proceed with our proof of the claim by cases. One case is left for you to do as an exercise.

Assume $n \in \mathbb{N}$. # n is a typical element of $\mathbb{N}$
Then $\exists k \in \mathbb{N}, n = 3k \vee n = 3k+1 \vee n = 3k+2$. # by (*)
Let $k_0 \in \mathbb{N}$ be such that $n = 3k_0 \vee n = 3k_0 + 1 \vee n = 3k_0 + 2$.
Case 1: Assume $n = 3k_0$.
Then $3(3k_0^2) = 9k_0^2 = n^2$. # algebra
Then $\exists k \in \mathbb{N}, n^2 = 3k$. # $k = 3k_0^2 \in \mathbb{N}$
Case 2: Assume $n = 3k_0 + 1$.
Then $3(3k_0^2 + 2k_0) + 1 = 9k_0^2 + 6k + 1 = n^2$. # algebra
Then $\exists k \in \mathbb{N}, n^2 = 3k + 1$. # $k = 3k_0^2 + 2k_0 \in \mathbb{N}$
Case 3: Assume $n = 3k_0 + 2$.
(Exercise.)
Then $(\exists k \in \mathbb{N}, n^2 = 3k) \vee (\exists k \in \mathbb{N}, n^2 = 3k+1)$. # true in all possible cases
Then $\forall n \in \mathbb{N}, (\exists k \in \mathbb{N}, n^2 = 3k) \vee (\exists k \in \mathbb{N}, n^2 = 3k+1)$. # introduction of universal

3.14 Building formulae and taking formulae apart

So far we've been concentrating on proving more and more complicated sentences. This makes sense, since the sentence we've proving determines the structure our proof will take. For each of the logical connectives and quantifiers, we've seen structures that allow us to conclude big statements from smaller ones. The inference rules that allow us to do this are collectively called INTRODUCTION RULES, since they allow us to introduce new sentences of a particular type.

But rarely do we prove things directly from predicates. We often have to use known theorems and results or separately proven lemmas to reduce the length of our proofs to a manageable size (can you imagine always having to prove $2 + 2 = 4$ from primitive sets each time you use this fact?). Good theorems are useful in a number of settings, and typically use a number of connectives and quantifiers. Knowing how to break complex sentences down is equally important as knowing how to build complex sentences up.

Just as there are inference rules allowing us to introduce new, complex sentences, there are inference rules allowing us to break sentences down in a formal, precise and valid way. These rules are collectively called ELIMINATION RULES, since they allow us to eliminate connectives and quantifiers we don't want anymore. Most rules should be fairly straight-forward and should make sense to you at this point; if not, you should review your manipulation rules.

Double negation elimination

We can't do much to remove one negation (unless we can move it further inside), but we know how to get rid of two negations. Indeed, this was a manipulation rule from the previous chapter, but we can also treat it as a reasoning rule: if we know $\neg\neg A$ is true, we know A is true.

Conjunction elimination

Nearly as easy as negation, how can we break up a conjunction? If we know $A \wedge B$, what can we conclude?[13]

Existential elimination (or instantiation)

We might know that $\exists x \in D, B$, where B likely mentions x somewhere inside. In other words, we know B is true for some element in D, but we don't know which one. How can we proceed? We'd probably like to say something about that element in D that B is true for, but how do we know which element it is?

We don't really need to know which element B is true for, only that it exists. We can proceed by using B with every reference to x replaced by a new variable x' (just notation to distinguish it from x).

Disjunction elimination

$A \vee B$ itself cannot be split, as we don't know which part of the disjunction is true. However, if we also know $\neg A$, we can conclude B must be true. Analogously, with $\neg B$ we can conclude A.

Another good way to deal with a disjunction is proof by cases, which we discussed above.

Implication elimination

Suppose we know $A \Rightarrow B$. If we are able to show A is true, then we could immediately conclude B. This is perhaps the most basic reasoning structure, and has a fancy latin name: *modus ponens* (meaning "mode that affirms"). This form is the basis to deductive argument (you can imagine Sherlock Holmes using modus ponens to reveal the criminal).

On the other hand, if we knew $\neg B$, we could still get something from $A \Rightarrow B$: we'd be able to conclude $\neg A$. This form of reasoning is using the contrapositive and is known as *modus tollens* (Latin for "mode that denies").

We can also appeal to the manipulation rules to rewrite $A \Rightarrow B$ as a disjunction, $\neg A \vee B$, and expand this formula as desired.

Bi-implication elimination

To take apart a sentence like $A \Leftrightarrow B$, we simply exploit its equivalence to $(A \Rightarrow B) \wedge (B \Rightarrow A)$ and expand it appropriately.

If we also know A, we can skip some work and directly conclude that B must be true (using the implication $A \Rightarrow B$ hidden in the bi-implication). Likewise, if we also knew $\neg A$, we could conclude $\neg B$. Each of these properties are easily proven using preceding rules.

Universal elimination (or instantiation)

Suppose you know that $\forall x \in D, B(x)$. How can we use this fact to help prove other things? This sentence says $B(x)$ is true for all members of domain D. So we could use this as meaning a huge conjunction over all the elements of $D = \{d_1, d_2, d_3, \ldots\}$:

$$B(d_1) \wedge B(d_2) \wedge B(d_3) \wedge \ldots$$

From this expansion (even if we can't write it[14]) it's clear that if $a \in D$, we can conclude that $B(a)$ is true. This is sometimes called universal instantiation, or universal specialization, since we're allowed to conclude a specialized statement from our general statement. Intuitively, what holds for everything must hold for any specific thing. Typically, a will have been mentioned already, and you'll want to express that a has some specific property (in this case, $B(a)$).

3.15 Summary of inference rules

There are several basic and derived rules we're allowed to use in our proofs. Most of them are summarized below. For each rule, if you know (have already shown) everything that is above the line, you are allowed to conclude anything that's below the line.

Introduction rules

[$\neg$I] negation introduction

Assume A
$\vdots$
contradiction
$\neg A$

[$\wedge$I] conjunction introduction

A
B
$A \wedge B$

[$\vee$I] disjunction introduction

A
$A \vee B$
$B \vee A$

$A \vee \neg A$

[$\Rightarrow$I] implication introduction

(direct)
Assume A
$\vdots$
B
$A \Rightarrow B$

(indirect)
Assume $\neg B$
$\vdots$
$\neg A$
$A \Rightarrow B$

[$\Leftrightarrow$I] equivalence/bi-implication introduction

$A \Rightarrow B$
$B \Rightarrow A$
$A \Leftrightarrow B$

[$\forall$I] universal introduction

Assume $a \in D$
$\vdots$
$P(a)$
$\forall x \in D, P(x)$

[$\exists$I] existential introduction

$P(a)$
$a \in D$
$\exists x \in D, P(x)$

Elimination rules

[$\neg$E] negation elimination

$\neg\neg A$
A

A
$\neg A$
contradiction

[$\wedge$E] conjunction elimination

$A \wedge B$
A
B

[$\vee$E] disjunction elimination

$A \vee B$
$\neg A$
B

$A \vee B$
$\neg B$
A

[$\Rightarrow$E] implication elimination

(Modus Ponens)
$A \Rightarrow B$
A
B

(Modus Tollens)
$A \Rightarrow B$
$\neg B$
$\neg A$

[$\Leftrightarrow$E] equivalence/bi-implication elimination

$A \Leftrightarrow B$
$A \Rightarrow B$
$B \Rightarrow A$

[$\forall$E] universal elimination

$\forall x \in D, P(x)$
$a \in D$
$P(a)$

[$\exists$E] existential elimination

$\exists x \in D, P(x)$
Let $a \in D$ such that $P(a)$
$\vdots$

It may surprise you to learn that by this point, we've covered all of the basic proof techniques you will need during your undergraduate career (and beyond). There is one basic proof technique that we have yet to

cover (mathematical induction) but it is more properly the main subject of the course CSC 236 H. (Though we will discuss it a little bit in the next chapter.)

Given this, you may feel that the proofs we've worked on so far have been nowhere near as complicated as what you might find in your calculus textbook, for example. But if you take the time to examine the structure of any such proof, you will most likely find that all of the techniques it uses were covered in this chapter. The complexity of these proofs stem, not from using more complex techniques, but from their scale and their reliance on numerous other results and complex definitions.

This is no different from the contrast between small programs and large ones: both are written using the same programming language, which provides just a small set of "building blocks" — conditionals, loops, functions, *etc.* The complexity of larger programs stems mainly from their size and/or their reliance on numerous external libraries.

Bearing this in mind, you now have all of the tools required to understand and appreciate some of the deepest and most beautiful results in the theory of computation (see Chapter 5). But first, in the next chapter, we'll apply those tools to something more concrete: the analysis of algorithms.

Chapter 3 Notes

[1]Here's an example of a conjecture whose proof has evaded the best minds for almost 75 years — maybe you'll prove it? Define $f(n)$, for $n \in \mathbb{N}$, as follows:

$$f(n) = \begin{cases} n/2, & n \text{ even}, \\ 3n+1, & n \text{ odd}. \end{cases}$$

Then define $f^{k+1}(n)$ as $f(f^k(n))$, for all $k \in \mathbb{N}$, with special case $f^0(n) = n$. (So $f^1(n) = f(n)$, $f^2(n) = f(f(n))$, *etc.*)
Conjecture: $\forall n \in \mathbb{N}, n \geqslant 1 \Rightarrow \exists k \in \mathbb{N}, f^k(n) = 1$.
Easy to state, but (so far) hard to prove or disprove.

[2]For example, "for any two points on the plane, there is exactly one line that passes through both points" is an axiom in Euclidian geometry.

[3]The contrapositive.

[4]$\forall x \in \mathbb{R},\ 1/(x+2) < 3 \Rightarrow x > 0$. False, for example let $x = -4$, then $1/(-4+2) = -1/2 < 3$ but $-4 \not> 0$. Indeed, every $x < -2$ is a counter-example.

[5] Assume $e \in \mathbb{R}$. # typical element of $\mathbb{R}$
Assume $e > 0$. # antecedent
Let $d_e = \ldots$ # something helpful, probably depending on e
Then $d_e \in \mathbb{R}$. # verify d_e is in the domain
Then $d_e > 0$. # show d_e is positive
Assume $x \in \mathbb{R}$. # typical element of $\mathbb{R}$
Assume $0 < |x - a| < d_e$. # antecedent
$\vdots$
Then $|f(x) - l| < e$. # inner consequent
Then $0 < |x - a| < d_e \Rightarrow (|f(x) - l| < e)$. # introduce implication
Then $\forall x \in \mathbb{R},\ 0 < |x - a| < d_e \Rightarrow (|f(x) - l| < e)$. # introduce universal
Then $\exists d \in \mathbb{R}, d > 0 \wedge (\forall x \in \mathbb{R},\ 0 < |x - a| < d \Rightarrow (|f(x) - l| < e))$. # introduce existential
Then, $e > 0 \Rightarrow (\exists d \in \mathbb{R}, d > 0 \wedge (\forall x \in \mathbb{R},\ 0 < |x - a| < d \Rightarrow (|f(x) - l| < e)))$.
Then $\forall e \in \mathbb{R},\ e > 0 \Rightarrow (\exists d \in \mathbb{R}, d > 0 \wedge (\forall x \in \mathbb{R},\ 0 < |x - a| < d \Rightarrow (|f(x) - l| < e)))$.

[6]In other words, a formula that depends only on i. In this case, we see that $a_i = i^2$.

[7]We need to prove both pieces of a conjunction.

[8]Try $j = i + 2$.

[9]$\forall a \in \mathbb{N}, \forall b \in \mathbb{N}, b > 0 \Rightarrow a + b > a$.

[10] Assume $x \in \mathbb{R}$.
⋮
Then $\lfloor x \rfloor < x + 1$.
Then $\forall x \in \mathbb{R}$, $\lfloor x \rfloor < x + 1$.

[11] Assume $x \in \mathbb{Z}$. # x is a typical integer
Either x is even or x is odd.
Case 1: [Assume] x is even.
Then $x(x + 1)$ is even. # if x is a multiple of 2, so is $x(x + 1)$
Case 2: [Assume] x is odd.
Then $x + 1$ is even. # if x leaves remainder 1, $x + 1$ leaves remainder 0
Then $x(x + 1)$ is even. # if $x + 1$ is a multiple of 2, so is $x(x + 1)$
Then $x(x + 1)$ is even. # true in all (both) possible cases
Then $\forall x \in \mathbb{Z}, x(x + 1)$ is even. # introduce universal

[12]Instead of concluding R in one case and S in the other, we are actually concluding $R \vee S$ in both cases, and then we bring $R \vee S$ outside the cases because we concluded it in each case, and one of the cases must hold. (Remember that once we conclude that R is true, we can immediately conclude that $R \vee S$ is true.) So this is exactly the same structure we've seen before.

[13]We know that A is true and that B is true.

[14]All our sentences are finite in length, so if our domain D is infinite (like the natural numbers or real numbers), we can't actually write this expansion down. That's the reason why we need a universal quantifier in our logic system.

Chapter 4

Algorithm Analysis and Asymptotic Notation

4.1 Correctness, running time of programs

So far we have been proving statements about databases, mathematics and arithmetic, or sequences of numbers. Though these types of statements are common in computer science, you'll probably encounter algorithms most of the time. Often we want to reason about algorithms and even prove things about them. Wouldn't it be nice to be able to *prove* that your program is correct? Especially if you're programming a heart monitor or a NASA spacecraft?

In this chapter we'll introduce a number of tools for dealing with computer algorithms, formalizing their expression, and techniques for analyzing properties of algorithms, so that we can prove correctness or prove bounds on the resources that are required.

4.2 Binary (base 2) notation

Let's first think about numbers. In our everyday life, we write numbers in decimal (base 10) notation (although I heard of one kid who learned to use the fingers of her left hand to count from 0 to 31 in base 2). In decimal, the sequence of digits 20395 represents (parsing from the right):

$$5 + 9(10) + 3(100) + 0(1000) + 2(10000) =$$
$$5(10^0) + 9(10^1) + 3(10^2) + 0(10^3) + 2(10^4)$$

Each position represents a power of 10, and 10 is called the base. Each position has a digit from $[0, 9]$ representing how many of that power to add. Why do we use 10? Perhaps due to having 10 fingers (however, humans at various times have used base 60, base 20, and mixed base 20,18 (Mayans)). In the last case there were $(105)_{20,18}$ days in the year. Any integer with absolute value greater than 1 will work (so experiment with base -2).

Consider using 2 as the base for our notation. What digits should we use?[1] We don't need digits 2 or higher, since they are expressed by choosing a different position for our digits (just as in base 10, where there is no single digit for numbers 10 and greater).
Here are some examples of binary numbers:

$$(10011)_2$$

represents

$$1(2^0) + 1(2^1) + 0(2^2) + 0(2^3) + 1(2^4) = (19)_{10}$$

We can extend the idea, and imitate the decimal point (with a "binary point"?) from base 10:

$$(1011.101)_2 = 19\frac{5}{8}$$

How did we do that?[2] Here are some questions:

- How do you multiply two base 10 numbers?[3] Work out 37×43.
- How do you multiply two binary numbers?[4]
- What does "right shifting" (eliminating the right-most digit) do in base 10?[5]
- What does "right shifting" do in binary?[6]
- What does the rightmost digit tell us in base 10? In binary?

Convert some numbers from decimal to binary notation. Try 57. We'd like to represent 57 by adding either 0 or 1 of each power of 2 that is no greater than 57. So $57 = 32 + 16 + 8 + 1 = (111001)_2$. We can also fill in the binary digits, systematically, from the bottom up, using the % operator from Python (the remainder after division operator, at least for positive arguments):

$$\begin{aligned} 57 \,\%\, 2 = 1 \quad &\text{so} \quad (?????1)_2 \\ (57-1)/2 = 28 \,\%\, 2 = 0 \quad &\text{so} \quad (????01)_2 \\ 28/2 = 14 \,\%\, 2 = 0 \quad &\text{so} \quad (???001)_2 \\ 14/2 = 7 \,\%\, 2 = 1 \quad &\text{so} \quad (??1001)_2 \\ (7-1)/2 = 3 \,\%\, 2 = 1 \quad &\text{so} \quad (?11001)_2 \\ (3-1)/2 = 1 \,\%\, 2 = 1 \quad &\text{so} \quad (111001)_2 \end{aligned}$$

Addition in binary is the same as (only different from...) addition in decimal. Just remember that $(1)_2 + (1)_2 = (10)_2$. If we add two binary numbers, this tells us when to "carry" 1:

$$\begin{array}{rr} & 1011 \\ + & 1011 \\ \hline & 10110 \end{array}$$

LOG$_2$

How many 5-digit binary numbers are there (including those with leading 0s)? These numbers run from $(00000)_2$ through $(11111)_2$, or 0 through 31 in decimal — 32 numbers. Another way to count them is to consider that there are two choices for each digit, hence 2^5 strings of digits. If we add one more digit we get twice as many numbers. Every digit doubles the range of numbers, so there are two 1-digit binary numbers (0 and 1), four 2-digit binary numbers (0 through 3), 8 3-digit binary numbers (0 through 7), and so on.

Reverse the question: how many digits are required to represent a given number. In other words, what is the smallest integer power of 2 needed to exceed a given number? $\log_2 x$ is the power of 2 that gives $2^{\log_2 x} = x$. You can think of it as how many times you must multiply 1 by 2 to get x, or roughly the number of digits in the binary representation of x. (The precise number of digits needed is $\lceil \log_2(x+1) \rceil$ — which happens to be equal to $\lfloor (\log_2 x) + 1 \rfloor$ for all positive values of x).

4.3 LOOP INVARIANT FOR BASE 2 MULTIPLICATION

Integers are naturally represented on a computer in binary, since a gate can be in either an on or off (1 or 0) position. It is very easy to multiply or divide by 2, since all we need to do is perform a left or right shift (an easy hardware operation). Similarly, it is also very easy to determine whether an integer is even or odd. Putting these together, we can write a multiplication algorithm that uses these fast operations:

```
def mult(m,n):
    """ Multiply integers m and n. """
    # Precondition: m >= 0
    x = m
    y = n
    z = 0

    # loop invariant: z = mn - xy
    while x != 0:
        if x % 2 == 1:  z = z + y  # x is odd
        x = x >> 1  # x = x / 2 (right shift)
        y = y << 1  # y = y * 2 (left shift)

    # post condition: z = mn
    return z
```

After reading this algorithm, there is no reason you should believe it actually multiplies two integers: we'll need to prove it to you. Let's consider the precondition first. So long as m is a non-negative natural number, and n is an integer, the program claims to work. The postcondition states that z, the value that is returned, is equal to the product of m and n (that would be nice, but we're not convinced).

Let's look at the stated loop invariant. A LOOP INVARIANT is a relationship between the variables that is always true at the start and at the end of a loop iteration (we'll need to prove this). It's sufficient to verify that the invariant is true at the start of the first iteration, and verify that if the invariant is true at the start of any iteration, it must be true at the end of the iteration. Before we start the loop, we set $x = m$, $y = n$ and $z = 0$, so it is clear that $z = mn - xy = mn - mn = 0$. Now we need to show that if $z = mn - xy$ before executing the body of the loop, and $x \neq 0$, then after executing the loop body, $z = mn - xy$ is still true (can you write this statement formally?). Here's a sketch of a proof:

> Assume $x_i, y_i, z_i, x_{i+1}, y_{i+1}, z_{i+1}, m, n \in \mathbb{Z}$, where x_i represents the value of variable x at the beginning of the ith iteration of the loop, and similarly for the other variables and subscripts. (Note that there is no need to subscript m, n, since they aren't changed by the loop.)
>
> Assume $z_i = mn - x_i y_i$.
>
> Case 1: Assume x_i odd.
>
> Then $z_{i+1} = z_i + y_i$, $x_{i+1} = (x_i - 1)/2$, and $y_{i+1} = 2y_i$.
>
> So $mn - x_{i+1}y_{i+1} = mn - (x_i - 1)/2 \cdot 2y_i$ (since x_i is odd)
>
> $= mn - x_i y_i + y_i$
>
> $= z_i + y_i$
>
> $= z_{i+1}$.
>
> Case 2: Assume x_i even.
>
> Then $z_{i+1} = z_i$, $x_{i+1} = x_i/2$, and $y_{i+1} = 2y_i$.
>
> So $mn - x_{i+1}y_{i+1} = mn - x_i/2 \cdot 2y_i$
>
> $= mn - x_i y_i$
>
> $= z_i$
>
> $= z_{i+1}$.
>
> Since x_i is either even or odd, in all cases $mn - x_{i+1}y_{i+1} = z_{i+1}$
>
> Thus $mn - x_i y_i = z_i \Rightarrow mn - x_{i+1}y_{i+1} = z_{i+1}$.
>
> Since $x_i, x_{i+1}, y_i, y_{i+1}, z_i, z_{i+1}, m, n$ are arbitrary elements,
>
> $\forall x_i, x_{i+1}, y_i, y_{i+1}, z_i, z_{i+1}, m, n \in \mathbb{Z}, mn - x_i y_i = z_i \Rightarrow mn - x_{i+1}y_{i+1} = z_{i+1}$.

We should probably verify the postcondition to fully convince ourselves of the correctness of this algorithm. We've shown the loop invariant holds, so let's see what we can conclude when the loop terminates (*i.e.*,

when $x = 0$). By the loop invariant, $z = mn - xy = mn - 0 = mn$, so we know we must get the right answer (assuming the loop eventually terminates).

We should now be fairly convinced that this algorithm is in fact correct. One might now wonder, how many iterations of the loop are completed before the answer is returned?

Also, why is it necessary for $m \geqslant 0$? What happens if it isn't?

4.4 RUNNING TIME OF PROGRAMS

For any program P and any input x, let $t_P(x)$ denote the number of "steps" P takes on input x. We need to specify what we mean by a "step." A "step" typically corresponds to machine instructions being executed, or some indication of time or resources expended.
Consider the following (somewhat arbitrary) accounting for common program steps:

METHOD CALL: 1 step + steps to evaluate each argument + steps to execute the method.

RETURN STATEMENT: 1 step + steps to evaluate return value.

IF STATEMENT: 1 step + steps to evaluate condition.

ASSIGNMENT STATEMENT: 1 step + steps to evaluate each side.

ARITHMETIC, COMPARISON, BOOLEAN OPERATORS: 1 step + steps to evaluate each operand.

ARRAY ACCESS: 1 step + steps to evaluate index.

MEMBER ACCESS: 2 steps.

CONSTANT, VARIABLE EVALUATION: 1 step.

Notice that none of these "steps" (except for method calls) depend on the size of the input (sometimes denoted with the symbol n). The smallest and largest steps above differ from each other by a constant of about 5, so we can make the additional simplifying assumption that they all have the same cost — 1.

4.5 LINEAR SEARCH

Let's use linear search as an example.

```
def LS(A,x):
    """ Return an index i such that x == L[i].  Otherwise, return -1. """
    i = 0                # (line 1)
    while i < len(A):    # (line 2)
        if A[i] == x:    # (line 3)
            return i     # (line 4)
        i = i + 1        # (line 5)
    return -1            # (line 6)
```

Let's trace a function call, `LS([2,4,6,8],4)`:

Line 1: 1 step (`i=0`)

Line 2: 1 step (`0 < 4`)

Line 3: 1 step (`A[0] == 4`)

Line 5: 1 step (`i = 1`)

Line 2: 1 step (`1 < 4`)

Line 3: 1 step (`A[1] == 4`)

Line 4: 1 (`return 1`)

So $t_{LS}([2,4,6,8],4) = 7$. Notice that if the first index where x is found is j, then $t_{LS}(A,x)$ will count lines 2, 3, and 5 once for each index from 0 to $j-1$ (j indices), and then count lines 2, 3, 4 for index j, and so $t_{LS}(A,x)$ will be $1+3j+3$.

If x does not appear in A, then $t_{LS}(A,x) = 1+3\,\text{len}(A)+2$, because line 1 executes once, lines 2, 3, and 5 executes once for each index from 0 to $\text{len}(A)-1$, and then lines 2 and 6 execute.

We want a measure that depends on the size of the input, not the particular input. There are three standard ways. Let P be a program, and let I be the set of all inputs for P. Then:

Average-case complexity: the weighted average over all possible inputs of size n.

In general

$$A_P(n) = \sum_{x \text{ of size } n} t_P(x) \cdot p(x)$$

where $p(x)$ is the probability that input x is encountered.

Assuming all the inputs are equally likely, this "simplifies" to

$$A_P(n) = \frac{\sum_{x \text{ of size } n} t_P(x)}{\text{number of inputs of size } n}$$

(Difficult to compute.)

Best-case complexity: $\min(t_P(x))$, where x is an input of size n.

In other words, $B_P(n) = \min\{t_P(x) \mid x \in I \wedge \text{size}(x) = n\}$.

(Mostly useless.)

Worst-case complexity: $\max(t_P(x))$, where x is an input of size n.

In other words, $W_P(n) = \max\{t_P(x) \mid x \in I \wedge \text{size}(x) = n\}$.

(Relatively easy to compute, and gives a performance guarantee.)

What is meant by "input size"? This depends on the algorithm. For linear search, the number of elements in the array is a reasonable parameter. Technically (in csc 463 h, for example), the size is the number of bits required to represent the input in binary. In practice we use the number of elements of input (length of array, number of nodes in a tree, *etc.*)

4.6 Run time and constant factors

When calculating the running time of a program, we may know how many basic "steps" it takes as a function of input size, but we may not know how long each step takes on a particular computer. We would like to estimate the overall running time of an algorithm while ignoring constant factors (like how fast the CPU is). So, for example, if we have 3 machines, where operations take 3μs, 8μs and 0.5μs, the three functions measuring the amount of time required, $t(n) = 3n^2$, $t(n) = 8n^2$, and $t(n) = n^2/2$ are considered the same, ignoring ("to within") constant factors (the time required always grows according to a quadratic function in terms of the size of the input n).

To view this another way, think back to the linear search example in the previous section. The worst-case running time for that algorithm was given by the function

$$W(n) = 3n + 3$$

But what exactly does the constant "3" in front of the "n" represent? Or the additive "$+3$"? Neither value corresponds to any intrinsic property of the algorithm itself; rather, the values are consequences of some arbitrary choices on our part, namely, how many "steps" to count for certain Python statements. Someone counting differently (*e.g.*, counting more than 1 step for statements that access list elements by index) would arrive at a different expression for the worst-case running time. Would their answer be "more" or "less" correct than ours? Neither: both answers are just as imprecise as one another! This is why we want to come up with a tool that allows us to work with functions while ignoring constant multipliers.

The nice thing is that this means that lower order terms can be ignored as well! So $f(n) = 3n^2$ and $g(n) = 3n^2 + 2$ are considered "the same," as are $h(n) = 3n^2 + 2n$ and $j(n) = 5n^2$. Notice that

$$\forall n \in \mathbb{N}, n \geqslant 1 \Rightarrow f(n) \leqslant g(n) \leqslant h(n) \leqslant j(n)$$

but there's always a constant factor that can reverse any of these inequalities.

Really what we want to measure is the growth rate of functions (and in computer science, the growth rate of functions that bound the running time of algorithms). You might be familiar with binary search and linear search (two algorithms for searching for a value in a sorted array). Suppose one computer runs binary search and one computer runs linear search. Which computer will give an answer first, assuming the two computers run at roughly the same CPU speed? What if one computer is much faster (in terms of CPU speed) than the other, does it affect your answer? What if the array is really, really big?

How large is "sufficiently large?"

Is binary search a better algorithm than linear search?[7] It depends on the size of the input. For example, suppose you established that linear search has complexity $L(n) = 3n$ and binary search has complexity $B(n) = 9 \log_2 n$. For the first few n, $L(n)$ is smaller than $B(n)$. However, certainly for $n > 10$, $B(n)$ is smaller, indicating less "work" for binary search.

When we say "large enough" n, we mean we are discussing the ASYMPTOTIC behaviour of the complexity function (*i.e.*, the behaviour as n grows toward infinity), and we are prepared to ignore the behaviour near the origin.

4.7 Asymptotic notation: Making Big-O precise

We define $\mathbb{R}^{\geqslant 0}$ as the set of nonnegative real numbers, and define $\mathbb{R}^+$ as the set of positive real numbers. Here is a precise definition of "The set of functions that are eventually no more than f, to within a constant factor":

DEFINITION: For any function $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$ (*i.e.*, any function mapping naturals to nonnegative reals), let

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

Saying $g \in \mathcal{O}(f)$ says that "g grows no faster than f" (or equivalently, "f is an upper bound for g"), so long as we modify our understanding of "growing no faster" and being an "upper bound" with the practice of ignoring constant factors. Now we can prove some theorems.

Suppose $g(n) = 3n^2 + 2$ and $f(n) = n^2$. Then $g \in \mathcal{O}(f)$. To be more precise, we need to prove the statement $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 3n^2 + 2 \leqslant cn^2$. It's enough to find some c and B that "work" in order to prove the theorem.

Finding c means finding a factor that will scale n^2 up to the size of $3n^2 + 2$. Setting $c = 3$ almost works, but there's that annoying additional term 2. Certainly $3n^2 + 2 < 4n^2$ so long as $n \geqslant 2$, since $n \geqslant 2 \Rightarrow n^2 > 2$. So pick $c = 4$ and $B = 2$ (other values also work, but we like the ones we thought of first). Now concoct a proof of

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 3n^2 + 2 \leqslant cn^2.$$

Let $c' = 4$ and $B' = 2$.
Then $c' \in \mathbb{R}^+$ and $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$. # direct proof for an arbitrary natural number
  Then $n^2 \geqslant B'^2 = 4$. # squaring is monotonic on natural numbers
  Then $n^2 \geqslant 2$.
  Then $3n^2 + n^2 \geqslant 3n^2 + 2$. # adding $3n^2$ to both sides of the inequality
  Then $3n^2 + 2 \leqslant 4n^2 = c'n^2$ # re-write
Then $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow 3n^2 + 2 \leqslant c'n^2$ # introduce $\forall$ and $\Rightarrow$
Then $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 3n^2 + 2 \leqslant cn^2$. # introduce $\exists$ (twice)

So, by definition, $g \in \mathcal{O}(f)$.

A more complex example

Let's prove that $2n^3 - 5n^4 + 7n^6$ is in $\mathcal{O}(n^2 - 4n^5 + 6n^8)$. We begin with:

Let $c' =$ ____. Then $c' \in \mathbb{R}^+$.
Let $B' =$ ____. Then $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$. # arbitrary natural number and antecedent
  Then $2n^3 - 5n^4 + 7n^6 \leqslant \ldots \leqslant c'(n^2 - 4n^5 + 6n^8)$.
Then $\forall n \in \mathbb{N}, n \geqslant Bi' \Rightarrow 2n^3 - 5n^4 + 7n^6 \leqslant c'(n^2 - 4n^5 + 6n^8)$. # introduce $\Rightarrow$ and $\forall$
Hence, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 2n^3 - 5n^4 + 7n^6 \leqslant c(n^2 - 4n^5 + 6n^8)$. # introduce $\exists$

To fill in the ... we try to form a chain of inequalities, working from both ends, simplifying the expressions:

$$
\begin{aligned}
2n^3 - 5n^4 + 7n^6 &\leqslant 2n^3 + 7n^6 && \text{(drop } -5n^4 \text{ because it doesn't help us in an important way)} \\
&\leqslant 2n^6 + 7n^6 && \text{(increase } n^3 \text{ to } n^6 \text{ because we have to handle } n^6 \text{ anyway)} \\
&= 9n^6 \\
&\leqslant 9n^8 && \text{(simpler to compare)} \\
&= 2(9/2)n^8 && \text{(get as close to form of the simplified end result: now choose } c' = 9/2) \\
&= 2cn^8 \\
&= c'(-4n^8 + 6n^8) && \text{(reading bottom up: decrease } -4n^5 \text{ to } -4n^8 \text{ because we have to handle } n^8 \text{ anyway)} \\
&\leqslant c'(-4n^5 + 6n^8) && \text{(reading bottom up: drop } n^2 \text{ because it doesn't help us in an important way)} \\
&\leqslant c'(n^2 - 4n^5 + 6n^8)
\end{aligned}
$$

We never needed to restrict n in any way beyond $n \in \mathbb{N}$ (which includes $n \geqslant 0$), so we can fill in $c' = 9/2$, $B' = 0$, and complete the proof.

To prove or not to prove...

What would it mean to prove $n^4 \notin \mathcal{O}(3n^2)$? More precisely, we have to prove the negation of the statement $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in N, n \geqslant B \Rightarrow n^4 \leqslant c3n^2$. Before you consider the proof structure that follows, you might find it useful to work out that negation.

Assume $c \in \mathbb{R}^+$ and $B \in \mathbb{N}$. # arbitrary positive real number and natural number
  Let $n_0 =$ ____.
  $\vdots$
  So $n_0 \in \mathbb{N}$.

$\vdots$
So $n_0 \geqslant B$.
$\vdots$
So $n_0^4 > c3n_0^2$.
Then $\forall c \in \mathbb{R}^+, \forall B \in \mathbb{N}, \exists n \in \mathbb{N}, n \geqslant B \wedge n^4 > c3n^2$.

Here's our chain of inequalities (the third $\vdots$):

$$\begin{aligned} \text{And } n_0^4 &\geqslant n_0^3 && \text{(don't need full power of } n_0^4\text{)} \\ &= n_0 \cdot n_0^2 && \text{(make form as close as possible)} \\ &> c \cdot 3n_0^2 && \text{(if we make } n_0 > 3c \text{ and } n_0 > 0\text{)} \end{aligned}$$

Now pick $n_0 = \max(B, \lceil 3c + 1 \rceil)$.

The first $\vdots$ is:

Since $c > 0$, $3c + 1 > 0$, so $\lceil 3c + 1 \rceil \in \mathbb{N}$.
Since $B \in \mathbb{N}$, $\max(B, \lceil 3c + 1 \rceil) \in \mathbb{N}$.

The second $\vdots$ is:

$\max(B, \lceil 3c + 1 \rceil) \geqslant B$.

We also note just before the chain of inequalities:

$n_0 = \max(B, \lceil 3c + 1 \rceil) \geqslant \lceil 3c + 1 \rceil \geqslant 3c + 1 > 3c$.

Some points to note are:

- Don't "solve" for n until you've made the form of the two sides as close as possible.
- You're not exactly solving for n: you are finding a condition of the form $n >$ __ that makes the desired inequality true. You might find yourself using the "max" function a lot.

ONE LAST EXAMPLE

Let $g(n) = 2^n$ and $f(n) = n$. I want to show that $g \notin \mathcal{O}(f)$.

Assume $c \in \mathbb{R}^+$, assume $B \in \mathbb{N}$. # arbitrary values
  Let $n_0 =$ ____
  ... Then $n_0 \in \mathbb{N}$.
  ... Then $n_0 \geqslant B$.
  ... Then $2^{n_0} > cn_0$.
Then $\forall c \in \mathbb{R}, \forall B \in \mathbb{N}, \exists n \in \mathbb{N}, n \geqslant B \wedge g(n) > cf(n)$. # introduce $\forall$

So, I can conclude that $g \notin \mathcal{O}(f)$.

The tricky part in this proof is to find the value of n_0. Unfortunately, there is no elementary way to do this — one of the easiest ways involves an old friend....

4.8 CALCULUS!

Intuitively, big-Oh notation expresses something about how two functions compare as n tends toward infinity. But we know of another mathematical notion that captures a similar (though not identical) idea: the concept of *limit*.

Let's work out how these two concepts are related. When we study whether or not $f \in \mathcal{O}(g)$ (for arbitrary functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$), we are working with the inequality "$f(n) \leqslant cg(n)$". As long as $g(n) \neq 0$, this is equivalent to studying the inequality "$f(n)/g(n) \leqslant c$". Intuitively, we would like to know how the function $f(n)/g(n)$ behaves as n tends toward infinity. This is exactly what limits express! More precisely, recall the following definition, for all $L \in \mathbb{R}^{\geqslant 0}$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \iff \forall \varepsilon \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant n_0 \Rightarrow L - \varepsilon < \frac{f(n)}{g(n)} < L + \varepsilon$$

and the following special case:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff \forall \varepsilon \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant n_0 \Rightarrow \frac{f(n)}{g(n)} > \varepsilon$$

Now suppose that $\lim_{n\to\infty} f(n)/g(n) = L$. Intuitively, this tells us that $f(n)/g(n) \approx L$, for n "large enough." In that case, $f(n) \approx Lg(n)$ for n large enough, so we should be able to prove that $f \in \mathcal{O}(g)$:

Assume $\lim_{n\to\infty} f(n)/g(n) = L$.
  Then $\exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant n_0 \Rightarrow L - 1 < f(n)/g(n) < L + 1$. # definition of limit for $\varepsilon = 1$
  Then $\exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant n_0 \Rightarrow f(n) \leqslant (L+1)g(n)$.
  Then $f \in \mathcal{O}(g)$. # definition of $\mathcal{O}$, with $B = n_0$ and $c = L + 1$
Hence, $\lim_{n\to\infty} f(n)/g(n) = L \Rightarrow f \in \mathcal{O}(g)$.

Note that limits are "stronger" than big-Oh: they express a more restrictive property of functions f and g. For example, $x^2 \sin(x) \in \mathcal{O}(x^2)$ even though $\lim_{x\to\infty} \frac{x^2 \sin(x)}{x^2}$ is undefined.

Wrapping it up

Getting back to our earlier example, we can now complete the proof. Recall that $g(n) = 2^n$ and $f(n) = n$. We rely on the fact that $\lim_{n\to\infty} 2^n/n = \infty$.[8]

Assume $c \in \mathbb{R}^+$, assume $B \in \mathbb{N}$. # arbitrary values
  Then $\exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant n_0 \Rightarrow 2^n/n > c$. # definition of $\lim_{n\to\infty} 2^n/n = \infty$ with $\varepsilon = c$
  Let n_0 be such that $\forall n \in \mathbb{N}, n \geqslant n_0 \Rightarrow 2^n/n > c$, and $n' = \max(B, n_0)$.
  Then $n' \in \mathbb{N}$.
  Then $n' \geqslant B$. # by definition of max
  Then $2^{n'} > cn'$ because $2^{n'}/n' > c$. # by the first line above, since $n' \geqslant n_0$
  Then $n' \geqslant B \wedge g(n') \geqslant cf(n')$. # introduce $\wedge$
  Then $\exists n \in \mathbb{N}, n \geqslant B \wedge g(n) \geqslant cf(n)$. # introduce $\exists$
Then $\forall c \in \mathbb{R}, \forall B \in \mathbb{N}, \exists n \in \mathbb{N}, n \geqslant B \wedge g(n) > cf(n)$. # introduce $\forall$

4.9 Other bounds

By analogy with $\mathcal{O}(f)$, consider two other definitions:

Definition: For any function $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, let

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^{\geqslant 0} \mid \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \geqslant cf(n)\}.$$

To say "$g \in \Omega(f)$" expresses the concept that "g grows at least as fast as f" (f is a lower bound on g).

Definition: For any function $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, let

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}^{\geqslant 0} \mid \exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)\}.$$

To say "$g \in \Theta(f)$" expresses the concept that "g grows at the same rate as f" (f is a tight bound for g, or f is both an upper bound and a lower bound on g).

SOME THEOREMS

Here are some general results that we now have the tools to prove.

- $f \in \mathcal{O}(f)$.
- $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$.
- $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$.
- $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$.

Test your intuition about Big-O by doing the "scratch work" to answer the following questions:

- Are there functions f, g such that $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$ but $f \neq g$?[9]
- Are there functions f, g such that $f \notin \mathcal{O}(g)$, and $g \notin \mathcal{O}(f)$?[10]

To show that $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$, we need to find a constant $c \in \mathbb{R}^+$ and a constant $B \in \mathbb{N}$, that satisfy:

$$\forall n \in \mathbb{N}, n \geqslant B \Rightarrow f(n) \leqslant ch(n).$$

Since we have constants that scale h to g and then g to f, it seems clear that we need their product to scale g to f. And if we take the maximum of the two starting points, we can't go wrong. Making this precise:

THEOREM 4.1: For any functions $f, g, h : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$.

PROOF:

Assume $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)$.
  So $f \in \mathcal{O}(g)$.
  So $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n > B \Rightarrow f(n) \leqslant cg(n)$. # by def'n of $f \in \mathcal{O}(g)$
  Let $c_g \in \mathbb{R}^+, B_g \in \mathbb{N}$ be such that $\forall n \in \mathbb{N}, n \geqslant B_g \Rightarrow f(n) \leqslant c_g g(n)$.
  So $g \in \mathcal{O}(h)$.
  So $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \leqslant ch(n)$. # by def'n of $g \in \mathcal{O}(h)$
  Let $c_h \in \mathbb{R}^+, B_h \in \mathbb{N}$ be such that $\forall n \in \mathbb{N}, n \geqslant B_h \Rightarrow g(n) \leqslant c_h h(n)$.
  Let $c' = c_g c_h$. Let $B' = \max(B_g, B_h)$.
  Then, $c' \in \mathbb{R}^+$ (because $c_g, c_h \in \mathbb{R}^+$) and $B' \in \mathbb{N}$ (because $B_g, B_h \in \mathbb{N}$).
  Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
    Then $n \geqslant B_h$ (by definition of max), so $g(n) \leqslant c_h h(n)$.
    Then $n \geqslant B_g$ (by definition of max), so $f(n) \leqslant c_g g(n) \leqslant c_g c_h h(n)$.
    So $f(n) \leqslant c'h(n)$.
  Hence, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow f(n) \leqslant c'h(n)$.
  Therefore, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow f(n) \leqslant ch(n)$.
  So $f \in \mathcal{O}(g)$, by definition.
So $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$.

To show that $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$, it is enough to note that the constant, c, for one direction is positive, so its reciprocal will work for the other direction.[11]

THEOREM 4.2: For any functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$.

PROOF:

$$\begin{aligned}
& g \in \Omega(f) \\
\iff & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \geqslant cf(n) && \text{(by definition)} \\
\iff & \exists c' \in \mathbb{R}^+, \exists B' \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B' \Rightarrow f(n) \leqslant c'g(n) && \text{(letting } c' = 1/c \text{ and } B' = B\text{)} \\
\iff & f \in \mathcal{O}(g) && \text{(by definition)}
\end{aligned}$$

To show $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$, it's really just a matter of unwrapping the definitions.

Theorem 4.3: For any functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$.

Proof:

$g \in \Theta(f)$
$\Leftrightarrow$ (by definition)
$\exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)$.
$\Leftrightarrow$ (combined inequality, and $B = \max(B_1, B_2)$)
$\big(\exists c_1 \in \mathbb{R}^+, \exists B_1 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B_1 \Rightarrow g(n) \geqslant c_1 f(n)\big) \wedge$
$\big(\exists c_2 \in \mathbb{R}^+, \exists B_2 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B_2 \Rightarrow g(n) \leqslant c_2 f(n)\big)$
$\Leftrightarrow$ (by definition)
$g \in \Omega(f) \wedge g \in \mathcal{O}(f)$

Here's an example of a corollary that recycles some of the theorems we've already proven (so we don't have to do the grubby work). To show $g \in \Theta(f) \Leftrightarrow f \in \Theta(g)$, I re-use theorems proved above and the commutativity of $\wedge$:

Corollary: For any functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $g \in \Theta(f) \Leftrightarrow f \in \Theta(g)$.

Proof:

$$
\begin{aligned}
& g \in \Theta(f) \\
\iff\ & g \in \mathcal{O}(f) \wedge g \in \Omega(f) && \text{(by 4.3)} \\
\iff\ & g \in \mathcal{O}(f) \wedge f \in \mathcal{O}(g) && \text{(by 4.2)} \\
\iff\ & f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) && \text{(by commutativity of } \wedge\text{)} \\
\iff\ & f \in \mathcal{O}(g) \wedge f \in \Omega(g) && \text{(by 4.2)} \\
\iff\ & f \in \Theta(g) && \text{(by 4.3)}
\end{aligned}
$$

4.10 Asymptotic notation and algorithm analysis

Note that asymptotic notation (the Big-$\mathcal{O}$, Big-Ω, and Big-Θ definitions) bound the asymptotic growth rates of *functions*, as n approaches infinity. Often in computer science we use this asymptotic notation to bound functions that express the running times of algorithms, perhaps in best case or in worst case. Asymptotic notation *does not* express or bound the worst case or best case running time directly, only the *functions* expressing these values.

This distinction is subtle, but crucial to understanding both running times and asymptotic notation. So when we say that U is an upper bound on the worst-case running time of some program P, we mean the following (using the notation introduced earlier: I is the set of all inputs for program P, $t_p(x)$ is the number of steps executed by P on input x, and $T_P(n)$ denotes the worst-case running time of P):

$$
\begin{aligned}
& T_P \in \mathcal{O}(U) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T_P(n) \leqslant cU(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \max\{t_P(x) \mid x \in I \wedge \text{size}(x) = n\} \leqslant cU(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \forall x \in I, \text{size}(x) = n \Rightarrow t_P(x) \leqslant cU(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall x \in I, \text{size}(x) \geqslant B \Rightarrow t_P(x) \leqslant cU(\text{size}(x))
\end{aligned}
$$

In other words, to show that $T_P \in \mathcal{O}(U(n))$, you need to find constants c and B and show that for an arbitrary input x of size n, P takes at most $c \cdot U(n)$ steps.

In the other direction, when we say that L is a lower bound on the worst-case running time of algorithm P, we mean:

$$\begin{aligned} & T_P \in \Omega(L) \\ \iff & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \max\{t_P(x) \mid x \in I \wedge \text{size}(x) = n\} \geqslant cL(n) \\ \iff & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \exists x \in I, \text{size}(x) = n \wedge t_P(x) \geqslant cL(n) \end{aligned}$$

In other words, to prove that $T_p \in \Omega(L)$, we have to find constants c, B and for arbitrary n, find an input x of size n, for which we can show that P takes at least $cL(n)$ steps on input x.

4.11 INSERTION SORT EXAMPLE

Here is an intuitive[12] sorting algorithm:

```
def IS(A):
""" Sort the elements of A in non-decreasing order. """
    i = 1                               # (line 1)
    while i < len(A):                   # (line 2)
        t = A[i]                        # (line 3)
        j = i                           # (line 4)
        while j > 0 and A[j-1] > t:     # (line 5)
            A[j] = A[j-1]               # (line 6)
            j = j-1                     # (line 7)
        A[j] = t                        # (line 8)
        i = i+1                         # (line 9)
```

Let's find an upper bound for $T_{IS}(n)$, the maximum number of steps to Insertion Sort an array of size n. We'll use the proof format to prove and find the bound simultaneously — during the course of the proof we can fill in the necessary values for c and B.
We show that $T_{IS}(n) \in \mathcal{O}(n^2)$ (where $n = \text{len}(A)$):

Let $c' =$ ____. Let $B' =$ ____.
Then $c' \in \mathbb{R}^+$ and $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$, A is an array of length n, and $n \geqslant B'$.
Then lines 5–7 execute at most $n - 1$ times, which we can overestimate at $3n$ steps, plus 1 step for the last loop test.
Then lines 2–9 take no more than $n(5 + 3n) + 1 = 5n + 3n^2 + 1$ steps.
So $3n^2 + 5n + 1 \leqslant c'n^2$ (fill in the values of c' and B' that makes this so — setting $c' = 9, B' = 1$ should do).
Since n is the length of an arbitrary array A, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow T_{IS}(n) \leqslant c'n^2$ (so long as $B' \geqslant 1$).
Since c' is a positive real number and B' is a natural number,
$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T_{IS}(n) \leqslant cn^2$.
So $T_{IS} \in \mathcal{O}(n^2)$ (by definition of $\mathcal{O}(n^2)$).

Similarly, we prove a lower bound. Specifically, $T_{IS} \in \Omega(n^2)$:

Let $c' =$ ____. Let $B' =$ ____.
Then $c' \in \mathbb{R}^+$ and $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
Let $A' = [n-1, \ldots, 1, 0]$ (notice that this means $n \geqslant 1$).
Then at any point during the outside loop, $A'[0..(i-1)]$ contains the same elements as before but sorted (*i.e.*, no element from $A'[(i+1)..(n-1)]$ has been examined yet).

Then the inner while loop makes i iterations, at a cost of 3 steps per iteration, plus 1 for the final loop check, since the value $A'[i]$ is less than all the values $A'[0..(i-1)]$, by construction of the array.
Then the inner loop makes strictly greater than $2i+1$, or greater than or equal to $2i+2.$, steps.
Then (since the outer loop varies from $i=1$ to $i=n-1$ and we have $n-1$ iterations of lines 3 and 4, plus one iteration of line 1), we have that $t_{IS}(n) \geqslant 1+3+5+\cdots+(2n-1)+(2n+1) = n^2$ (the sum of the first n odd numbers), so long as n is at least 4.
So there is some array A of size n such that $t_{IS}(A) \geqslant c'n^2$.
This means $T_{IS}(n) \geqslant c'n^2$ (setting $B'=4, c'=1$ will do).
Since n was an arbitrary natural number, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow T_{IS}(n) \geqslant c'n^2$.
Since $c' \in \mathbb{R}^+$ and B' is a natural number, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T_{IS}(n) \geqslant cn^2$.
So $T_{IS} \in \Omega(n^2)$ (by definition of $\Omega(n^2)$).

From these proofs, we conclude that $T_{IS} \in \Theta(n^2)$.

4.12 Of algorithms and stockbrokers

Suppose you have a list of integers, like $[3, -5, 7, 1, -2, 0, 3, -2, 1]$, and you wish to find the maximum sum of any slice of the list (in the Python sense of the term "slice"). For example, perhaps the integers represent changes in the price of shares of your favourite stock, and solving this problem would tell you the maximum profit that might be achieved by purchasing and selling the stock at the right times.

How can we solve this problem? One obvious solution is to examine every possible slice of the original list and to compute the sum of each one, keeping track of the maximum sum we encounter. If you implement this algorithm in Python — you should try it for yourself, it's an excellent way to practice writing loops! — you might end up with something like the following. (Note that the code below is not particularly idiomatic — for instance, we could have used `for`-loops instead of `while`-loops, and we could have called the `sum` built-in function rather than write a loop. But these are *cosmetic* differences: the code performs the same work, and the current version has the advantage that all of that work is explicit, making it easier to account for when we analyze the running time).

```
def max_sum(L):
    max = 0                          # line  1
    # To generate all non-empty slices [i:j] for list L, i must take on values
    # from 0 to len(L)-1, and j must take on values from i+1 to len(L).
    i = 0                            # line  2
    while i < len(L):                # line  3
        j = i + 1                    # line  4
        while j <= len(L):           # line  5
            # Compute the sum of L[i:j].
            sum = 0                  # line  6
            k = i                    # line  7
            while k < j:             # line  8
                sum = sum + L[k]     # line  9
                k = k + 1            # line 10
            # Update max if appropriate.
            if sum > max:            # line 11
                max = sum            # line 12
            j = j + 1                # line 13
        i = i + 1                    # line 14
    # At this point, we've examined every slice.
    return max                       # line 15
```

The correctness of this code should be fairly obvious from the comments, except perhaps for one thing: what if the input L is a list that contains only negative integers, like $[-2, -1, -2, -3]$? Shouldn't the maximum sum of a slice be some negative integer, like -1? So why do we initialize max to 0?[13]

Now, let's analyze the worst-case running time of max_sum. Intuitively, $T(n) \in \mathcal{O}(n^3)$ (where $n =$ len(L)) because of the triply-nested loops, each one of which iterates no more than n times. Is $T(n) \in \Omega(n^3)$? This is perhaps not so obvious, but we will show that it is indeed the case. To begin the analysis, we must decide exactly which operations to count and how much to count for each one. We must strike a balance: if our counting is too fine-grained, we risk getting bogged down in arithmetic that is irrelevant; if it is too coarse, we risk ignoring significant fractions of the work carried out by the algorithm. Given how the algorithm is written, a reasonable middle ground is to count one "step" for each line of code that is executed.

$T(n) \in \mathcal{O}(n^3)$

Analyzing the loops inside-out, we find that:

- The loop on lines 8–10 iterates $j - i$ times (once for each value of $k = i, i+1, \ldots, j-1$). And $j - i \leqslant n$ (since $j \leqslant n$ and $i \geqslant 0$). And the loop executes 3 steps at each iteration. So the innermost loop executes *no more* than $3n$ steps.
- The loop on lines 5–13 iterates $n - i$ times (once for each value of $j = i+1, i+2, \ldots, n$). And $n - i \leqslant n$ (since $i \geqslant 0$). And the loop executes at most $7 + 3n$ steps at each iteration: lines 5, 6, 7, 8, 11, 12, 13 each execute at most once, in addition to the $\leqslant 3n$ steps executed by the inner loop (line 5 always executes one more time than the number of iterations of the inner loop, when the loop condition becomes false; line 12 may or may not execute, but it certainly executes no more than once for each iteration). So the middle loop executes *at most* $7n + 3n^2$ steps.
- The loop on lines 3–14 iterates n times (once for each value of $i = 0, 1, \ldots, n-1$). And the loop executes at most $4 + 7n + 3n^2$ steps at each iteration: lines 3, 4, 5, 14 each execute at most once, in addition to the $\leqslant 7n + 3n^2$ steps executed by the middle loop. So the outer loop executes *no more* than $4n + 7n^2 + 3n^3$ steps.
- Lines 1, 2, 3, and 15 each execute once, so the algorithm executes $\leqslant 4 + 4n + 7n^2 + 3n^3$ steps in total, for every input list L.

Now we are ready to write the proof formally.

> Assume $n \in \mathbb{N}$ and $n \geqslant 1$ and L is a list with len(L) = n.
>
> As argued above, lines 8–10 perform at most $3n$ steps.
>
> Then lines 5–13 perform at most $(7 + 3n) \cdot n \leqslant 10n^2$ steps (as argued above).
>
> Then lines 3–14 perform at most $(4 + 10n^2) \cdot n \leqslant 14n^3$ steps (as argued above).
>
> Then the entire algorithm performs at most $4 + 14n^3 \leqslant 18n^3$ steps.
>
> Since n and L were arbitrary, $\forall n \in \mathbb{N}, n \geqslant 1 \Rightarrow T(n) \leqslant 18n^3$.
>
> Hence $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T(n) \leqslant cn^3$, *i.e.*, $T(n) \in \mathcal{O}(n^3)$.

(Note that this proof really is not complete without the rough work above, which we simply did not bother to repeat.)

$T(n) \in \Omega(n^3)$

The algorithm has three nested loops, so it's tempting to think that it is "obvious" that the running time is $\Omega(n^3)$. But consider this: there are many pairs i, j for which the loop for k performs few iterations (when $j - i$ is small), so we are over-counting when we say that the loop for k performs "at most" n iterations. The question is: by how much are we over-counting?

To show that $T(n) \in \Omega(n^3)$, we have to convince ourselves that the loop over k really does iterate at least some fraction of n times, for at least some fraction of n^2 many pairs i, j. There are many ways to do this; here is one that is relatively simple (splitting up the range of values for i, j, k evenly).

- The loop over i iterates at least $\lceil n/3 \rceil$ times — for each of the values $i = 0, 1, \ldots, \lceil n/3 \rceil$. (Really, we know it performs more than this many iterations, but since we're working on proving a lower bound it's okay to under-estimate here — we'll soon see why it makes sense to under-estimate in this way.)
- For each of these iterations of the outer loop i, the loop for j iterates at least $\lceil n/3 \rceil$ times — for each of the values $j = n - \lceil n/3 \rceil + 1, \ldots, n$. (Again, we know that the loop really performs more work than this but we're deliberately ignoring some of the work in order to guarantee a certain range of values for k, in the quest for our lower bound.)
- For each of these pairs i, j, the inner loop for k iterates over the values $i, i+1, \ldots, j-1$. This will always be at least $\lceil n/3 \rceil$ many iterations, since $i \leqslant \lceil n/3 \rceil$ and $j \geqslant n - \lceil n/3 \rceil + 1$. And each iteration of the innermost loop performs at least 1 step (more than that, really, but again we're under-estimating).

Formally, assume $n \in \mathbb{N}$ and $n \geqslant 3$.

Let $L = [1, 2, \ldots, n]$. Then L is a list of length n.

As argued above, the call `max_sum(L)` performs at least $\lceil n/3 \rceil \cdot \lceil n/3 \rceil \cdot \lceil n/3 \rceil \geqslant n^3/27$ many steps.

Since this happens specifically for list L, $T(n) \geqslant n^3/27$. # by definition of "worst-case"

Since n was arbitrary, $\forall n \in \mathbb{N}, n \geqslant 3 \Rightarrow T(n) \geqslant n^3/27$.

Hence $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T(n) \geqslant cn^3$, *i.e.*, $T(n) \in \Omega(n^3)$.

Doing better

If you examine the algorithm, you might notice one somewhat obvious inefficiency. In case you cannot see it, or to confirm that we're thinking of the same thing, try the following exercise: trace through the execution of the algorithm on input $L = [3, -5, 7, 1, -2, 0, 3, -2, 1]$, when $i = 1$ (with max $= 7$). You should notice that the algorithm computes sum $= L[1]$, then sum $= L[1] + L[2]$, *etc.* But each time, it starts over: for example, to compute $L[1] + L[2] + \cdots + L[5]$, the algorithm completely discards the previous value of sum (equal to $L[1] + L[2] + L[3] + L[4]$) and starts adding $L[1]$ and $L[2]$ and.... We could save computing time if we kept the old value and simply added $L[5]$ to it!

The general idea is to add values to a running total every time that the value of j changes, instead of having a separate inner loop. See if you can implement this change on your own. You might end up with something like the following.

```
def faster_max_sum(L):
    max = 0                          # line  1
    # Generate all non-empty slices [i:j+1] for list L, where i takes on values
    # from 0 to len(L)-1, and j takes on values from i to len(L)-1.
    i = 0                            # line  2
    while i < len(L):                # line  3
        sum = 0                      # line  4
        j = i                        # line  5
        while j < len(L):            # line  6
            sum = sum + L[j]         # line  7
            if sum > max:            # line  8
                max = sum            # line  9
            j = j + 1                # line 10
        i = i + 1                    # line 11
    # At this point, we've examined every slice.
    return max                       # line 12
```

How does this change the running time? You should be able to convince yourself that the new algorithm's running time is $\Theta(n^2)$ — a big improvement over the first algorithm. The details are left as an exercise.

Before we turn the page on this problem, you may wonder: is this the most efficient way to solve it? As it turns out, there is a clever algorithm that can figure out the maximum sum in worst-case time $\Theta(n)$ — in other words, with just one loop over the values! Rather than spoil the fun, we're going to let you puzzle this one out. . . .[14]

4.13 EXERCISES FOR ASYMPTOTIC NOTATION

1. Prove or disprove the following claims:
   (a) $7n^3 + 11n^2 + n \in \mathcal{O}(n^3)$[15]
   (b) $n^2 + 165 \in \Omega(n^4)$
   (c) $n! \in \mathcal{O}(n^n)$
   (d) $n \in \mathcal{O}(n \log_2 n)$
   (e) $\forall k \in \mathbb{N}, k > 1 \Rightarrow \log_k n \in \Theta(\log_2 n)$

2. Define $g(n) = \begin{cases} n^3/165, & n < 165 \\ \lceil \sqrt{6n^5} \rceil, & n \geqslant 165 \end{cases}$. Note that $\forall x \in \mathbb{R}, x \leqslant \lceil x \rceil < x + 1$.
   Prove that $g \in \mathcal{O}(n^{2.5})$.

3. Let $\mathcal{F}$ be the set of functions from $\mathbb{N}$ to $\mathbb{R}^{\geqslant 0}$. Prove the following theorems:
   (a) For $f, g \in \mathcal{F}$, if $g \in \Omega(f)$ then $g^2 \in \Omega(f^2)$.
   (b) $\forall k \in \mathbb{N}, k > 1 \Rightarrow \forall d \in \mathbb{R}^+, d \log_k n \in \Theta(\log_2 n)$.[16]

   Notice that (b) means that all logarithms eventually grow at the same rate (up to a multiplicative constant), so the base doesn't matter (and can be omitted inside the asymptotic notation).

4. Let $\mathcal{F}$ be the set of functions from $\mathbb{N}$ to $\mathbb{R}^{\geqslant 0}$. Prove or disprove the following claims:
   (a) $\forall f \in \mathcal{F}, \forall g \in \mathcal{F}, f \in \mathcal{O}(g) \Rightarrow (f + g) \in \Theta(g)$
   (b) $\forall f \in \mathcal{F}, \forall f' \in \mathcal{F}, \forall g \in \mathcal{F}, (f \in \mathcal{O}(g) \wedge f' \in \mathcal{O}(g)) \Rightarrow (f + f') \in \mathcal{O}(g)$

5. For each function f in the left column, choose one expression $\mathcal{O}(g)$ from the right column such that $f \in \mathcal{O}(g)$. Use each expression exactly once.

(i) $3 \cdot 2^n \in$ _____	(a) $\mathcal{O}(\frac{1}{n})$
(ii) $\frac{2n^4+1}{n^3+2n-1} \in$ _____	(b) $\mathcal{O}(1)$
(iii) $(n^5 + 7)(n^5 - 7) \in$ _____	(c) $\mathcal{O}(\log_2 n)$
(iv) $\frac{n^4 - n \log_2 n}{n^2+1} \in$ _____	(d) $\mathcal{O}(n)$
(v) $\frac{n \log_2 n}{n-5} \in$ _____	(e) $\mathcal{O}(n \log_2 n)$
(vi) $8 + \frac{1}{n^2} \in$ _____	(f) $\mathcal{O}(n^2)$
(vii) $2^{3n+1} \in$ _____	(g) $\mathcal{O}(n^{10})$
(viii) $n! \in$ _____	(h) $\mathcal{O}(2^n)$
(ix) $\frac{5 \log_2(n+1)}{1+n \log_2 3n} \in$ _____	(i) $\mathcal{O}(10^n)$
(x) $(n-2) \log_2(n^3 + 4) \in$ _____	(j) $\mathcal{O}(n^n)$

4.14 Exercises for algorithm analysis

1. Write a detailed analysis of the worst-case running time of algorithm `faster_max_sum`.

2. Write a detailed analysis of the worst-case running time of the following algorithm.

```
def mystery1(L):
""" L is a non-empty list of length len(L) = n. """
    tot = 0
    i = 0
    while i < len(L):
        if L[i] > 0:
            tot = tot + L[i]
        i = i + 1
    return tot
```

3. Write a detailed analysis of the worst-case running time of the following algorithm.

```
def mystery2(L):
""" L is a non-empty list of length len(L) = n. """
    i = 1
    while i < len(L) - 1:
        j = i - 1
        while j <= i + 1:
            L[j] = L[j] + L[i]
            j = j + 1
        i = i + 1
```

4. Write a detailed analysis of the worst-case running time of the following algorithm.

```
def mystery3(L):
""" L is a non-empty list of length len(L) = n. """
    i = 1
    while i < len(L):
        print L[i]
        i = i * 2
```

5. Write a detailed analysis of the worst-case running time of the following algorithm.

```
def mystery4(L):
""" L is a non-empty list of length len(L) = n. """
    i = 0
    while i < len(L):
        if L[i] % 2 == 0:
            j = i
            while j < len(L):
                L[j] = L[j] + 1
                j = j + 1
        i = i + 1
```

4.15 INDUCTION INTERLUDE

Suppose $P(n)$ is some predicate of the natural numbers, and:

$$(*) \quad P(0) \wedge (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)).$$

You should certainly be able to show that $(*)$ implies $P(0)$, $P(1)$, $P(2)$, in fact $P(n)$ where n is any natural number you have the patience to follow the chain of results to obtain. In fact, we feel that we can "turn the crank" enough times to show that $(*)$ implies $P(n)$ for any natural number n. This is called the Principle of Simple Induction (PSI). It isn't proved, it is an axiom that we assume to be true.
Here's an application of the PSI to some functions we've encountered before.

$P(n)$: $2^n \geqslant 2n$.

I'd like to prove that $\forall n, P(n)$, using the PSI. Here's what I do:

PROVE $P(0)$: $P(0)$ states that $2^0 = 1 \geqslant 2(0) = 0$, which is true.

PROVE $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$:
Assume $n \in \mathbb{N}$. # arbitrary natural number
Assume $P(n)$, that is $2^n \geqslant 2n$. # antecedent
Then $n = 0 \vee n > 0$. # natural numbers are non-negative
CASE 1 (assume $n = 0$): Then $2^{n+1} = 2^1 = 2 \geqslant 2(n+1) = 2$.
CASE 2 (assume $n > 0$): Then $n \geqslant 1$. # n is an integer greater than 0
Then $2^n \geqslant 2$. # since $n \geqslant 1$, and 2^n is monotone increasing
Then $2^{n+1} = 2^n + 2^n \geqslant 2n + 2 = 2(n+1)$. # by previous line and IH $P(n)$
Then $2^{n+1} \geqslant 2(n+1)$, which is $P(n+1)$. # true in both possible cases
Then $P(n) \Rightarrow P(n+1)$. # introduce $\Rightarrow$
Then $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$. # introduce $\forall$

I now conclude, by the PSI, $\forall n \in \mathbb{N}, P(n)$, that is $2^n \geqslant 2n$.

What happens to induction for predicates that are true for all natural numbers after a certain point, but untrue for the first few natural numbers? For example, 2^n grows much more quickly than n^2, but 2^3 is not larger than 3^2. Choose n big enough, though, and it is true that:

$$P(n) : 2^n > n^2.$$

You can't prove this for all n, when it is false for $n = 2, n = 3$, and $n = 4$, so you'll need to restrict the domain and prove that for all natural numbers greater than 4, $P(n)$ is true. We don't have a slick way to restrict domains in our symbolic notation. Let's consider three ways to restrict the natural numbers to just those greater than 4, and then use induction.

RESTRICT BY SET DIFFERENCE: One way to restrict the domain is by set difference:

$$\forall n \in \mathbb{N} \setminus \{0, 1, 2, 3, 4\}, P(n)$$

Again, we'll need to prove $P(5)$, and then that $\forall n \in \mathbb{N} \setminus \{0, 1, 2, 3, 4\}, P(n) \Rightarrow P(n+1)$.

RESTRICT BY TRANSLATION: We can also restrict the domain by translating our predicate, by letting $Q(n) = P(n+5)$, that is:

$$Q(n) : 2^{n+5} > (n+5)^2$$

Now our task is to prove $Q(0)$ is true and that for all $n \in \mathbb{N}$, $Q(n) \Rightarrow Q(n+1)$. This is simple induction.

Restrict using implication: Another method of restriction uses implication to restrict the domain where we claim $P(n)$ is true — in the same way as for sentences:

$$\forall n \in \mathbb{N}, n \geqslant 5 \Rightarrow P(n).$$

The expanded predicate $Q(n) : n \geqslant 5 \Rightarrow P(n)$ now fits our pattern for simple induction, and all we need to do is prove:

1. $Q(0)$ is true (it is vacuously true, since $0 \geqslant 5$ is false).
2. $\forall n \in \mathbb{N}, Q(n) \Rightarrow Q(n+1)$. This breaks into cases.
   - If $n < 4$, then $Q(n)$ and $Q(n+1)$ are both vacuously true (the antecedents of the implication are false, since n and $n+1$ are not greater than, nor equal to, 5), so there is nothing to prove.
   - If $n = 4$, then $Q(n)$ is vacuously true, but $Q(n+1)$ has a true antecedent ($5 \geqslant 5$), so we need to prove $Q(5)$ directly: $2^5 > 5^2$ is true, since $32 > 25$.
   - If $n > 4$, we can depend on the assumption of the consequent of $Q(n-1)$ being true to prove $Q(n)$:

$$\begin{aligned} 2^n &= 2^{n-1} + 2^{n-1} \quad \text{(definition of } 2^n\text{)} \\ &> 2(n-1)^2 \quad \text{(antecedent of } Q(n-1)\text{)} \\ &= 2n^2 - 2n + 2 = n^2 + n(n-2) + 2 \geqslant n^2 + 2 > n^2 \quad \text{(since } n > 4 \geqslant 2\text{)} \end{aligned}$$

After all that work, it turns out that we need prove just two things:

1. $P(5)$
2. $\forall n \in \mathbb{N}$, If $n > 4$, then $P(n) \Rightarrow P(n+1)$.

This is the same as before, except now our base case is $P(5)$ rather than $P(0)$, and we get to use the fact that $n \geqslant 5$ in our induction step (if we need it).

Whichever argument you're comfortable with, notice that simple induction is basically the same: you prove the base case (which may now be greater than 0), and you prove the induction step.

Chapter 4 Notes

[1] From 0 to $(2-1)$, if we work in analogy with base 10.

[2] To parse the 0.101 part, calculate $0.101 = 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3})$.

[3] You should be able to look up this algorithm in an elementary school textbook.

[4] Same as the previous exercise, but only write numbers that have 0's and 1's, and do binary addition.

[5] Integer divide by 10.

[6] Integer divide by 2.

[7] Better in the sense of time complexity.

[8] Applying l'Hôpital's Rule, $\lim_{n\to\infty} \frac{2^n}{n} = \lim_{n\to\infty} \frac{\ln(2)\cdot 2^n}{1} = \infty$, because $\lim_{n\to\infty} 2^n = \lim_{n\to\infty} n = \infty$.

[9] Sure, $f = n^2$, $g = 3n^2 + 2$.

[10]Sure. f and g don't need to both be monotonic, so let $f(n) = n^2$ and

$$g(n) = \begin{cases} n, & n \text{ even} \\ n^3, & n \text{ odd} \end{cases}$$

So not every pair of functions from $\mathbb{N} \to \mathbb{R}^{\geqslant 0}$ can be compared using Big-O.

[11]Let's try the symmetrical presentation of bi-implication.

[12]but not particularly efficient...

[13]Don't forget the *empty slice* $L[0:0]$ (or $L[i:i]$ for any index i, for that matter). This is a valid slice for any list and its sum is 0, which is greater than any negative integer.

[14]Hint: you don't need a loop to figure out the maximum sum of any slice that ends at index i.

[15]The claim is true.

Let $c' = 8$. Then $c' \in \mathbb{R}^+$.
Let $B' = 12$. Then $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
Then $n^3 = n \cdot n^2 \geqslant 12 \cdot n^2 = 11n^2 + n^2 \geqslant 11n^2 + n$. # since $n \geqslant B' = 12$
Thus $c'n^3 = 8n^3 = 7n^3 + n^3 \geqslant 7n^3 + 11n^2 + n$.
So $\forall n \in \mathbb{N},\ n \geqslant B' \Rightarrow 7n^3 + 11n^2 + n \leqslant c'n^3$.
Since B' is a natural number, $\exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 7n^3 + 11n^2 + n \leqslant c'n^3$.
Since c' is a real positive number, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 7n^3 + 11n^2 + n \leqslant cn^3$.
By definition, $7n^3 + 11n^2 + n \in \mathcal{O}(n^3)$.

[16] Assume $k \in \mathbb{N}$ and $k > 1$.
Assume $d \in \mathbb{R}^+$.
It suffices to argue that $d \log_k n \in \Theta(\log_2 n)$.
Let $c_1' = \frac{d}{\log_2 k}$. Since $k > 1$, $\log_2 k \neq 0$ and so $c_1' \in \mathbb{R}^+$.
Let $c_2' = \frac{d}{\log_2 k}$. By the same reasoning, $c_2' \in \mathbb{R}^+$.
Let $B' = 1$. Then $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
Then $c_1' \log_2 n = \frac{d}{\log_2 k} \log_2 n = d\frac{\log_2 n}{\log_2 k} = d \log_k n \leqslant d \log_k n$.
Moreover, $d \log_k n \leqslant d\frac{\log_2 n}{\log_2 k} = \frac{d}{\log_2 k} \log_2 n = c_2' \log_2 n$.
Hence, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow c_1' \log_2 n \leqslant d \log_k n \leqslant c_2' \log_2 n$.
Thus $\exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow c_1 \log_2 n \leqslant d \log_k n \leqslant c_2 \log_2 n$.
By definition, $d \log_k n \in \Theta(\log_2 n)$.
Thus, $\forall d \in \mathbb{R}^+, d \log_k n \in \Theta(\log_2 n)$.
Hence $\forall k \in \mathbb{N}, k > 1 \Rightarrow \forall d \in \mathbb{R}^+, d \log_k n \in \Theta(\log_2 n)$.

Chapter 5

A Taste of Computability Theory

5.1 The problem

Algorithms (implemented as computer programs) can carry out many complex tasks. Some of the more interesting ones concern computers and programs themselves, *e.g.*, compiling, interpreting, and other manipulations of source code.
Here is one particular task that we would like to carry out.

```
def halt(f,i):
    """Return True iff f(i) eventually halts."""

    return True  # replace this stub with correct code
```

Note that function `halt` is well-defined: `halt` is passed a reference to some other Python function `f` along with an input `i`, and it must return `True` if `f(i)` eventually halts (normally, or because of a crash); `False` if `f(i)` eventually gets stuck in some infinite loop. These are the only two possible behaviours for the call `f(i)` — we ignore any possibility of hardware failure (which could not be detected from within `halt` anyway). In other words, we are interested in the *conceptual* behaviour of `f` itself, under ideal conditions for its execution.

Before you read the next section, see if you can come up with an implementation that works — even if it's just at a high-level and not fully written out in Python. As a guide, think about the value returned by your code for the call `halt(blah,5)`, or for the call `halt(blah,8)`, where `blah` is the following function.

```
def blah(x):
    if x % 2 == 0:
        while True:  pass
    else
        return x
```

5.2 An impossible proof

What if we told you that it is impossible to implement function `halt`? Note that this is a very strong claim to make: we're NOT just saying "we don't know how to write `halt`"; we're saying "NOBODY can write `halt`, ever, because it simply cannot be done"!

Because this is such a strong claim, it is daunting to prove. Also, how can we even prove that something is *not possible*? Wouldn't that require us to argue about every possible way to try to carry out this task?[1]

Yet, we will be able to do just that using only the proof techniques we've seen so far. Here is how...

Assume `halt` exists (*i.e.*, it is fully written out in Python).
Then consider the following function.

```
def confused(f):
    def halt(f,i):
        ...copy/paste the full code for halt here...

    if halt(f,f):                # line 1
        while True:  pass        # line 2
    else:
        return False             # line 3
```

Note that this is a correct Python function (assuming we've pasted in the code for `halt`).[2]
Now, we ask: what is the behaviour of the call `confused(confused)`?
Either `confused(confused)` halts or `confused(confused)` does not halt.[3]
Case 1: Assume `confused(confused)` halts.
Then `halt(confused,confused)` returns `True` (on line 1). # by definition of `halt`
Then `confused(confused)` goes into an infinite loop (on line 2).
So `confused(confused)` halts $\Rightarrow$ `confused(confused)` does not halt.
Case 2: Assume `confused(confused)` does not halt.
Then `halt(confused,confused)` returns `False` (on line 1). # by definition of `halt`
Then `confused(confused)` returns `False` (on line 3).
So `confused(confused)` does not halt $\Rightarrow$ `confused(confused)` halts.
Hence, `confused(confused)` halts $\Leftrightarrow$ `confused(confused)` does not halt. # $\Leftrightarrow$I
Clearly, this is a contradiction. # of the form $p \Leftrightarrow \neg p$
Then, by contradiction, `halt` does not exist!

This is such a counter-intuitive result that it is tempting to dismiss it at first. But take the time to think through each step of the proof, and you will see that they are all correct. Moreover, this result does NOT expose a weakness in Python: the exact same argument would apply to every possible programming language (in fact, to things that we would not even call "programming languages").

HISTORICAL CONTEXT

The proof we just showed was discovered by the mathematicians Alan Turing and Alonzo Church (independently of each other), back in the late 1930's — before computers even existed! They argued that every possible algorithm can be expressed using a small set of primitive operations ("λ-calculus" in the case of Church and "Turing Machines" in the case of Turing). Then, they carried out the argument above based on those primitive operations.

If you believe that Python is capable of expressing every possible algorithm (and it is), then the argument shows that some problems cannot be solved by any algorithm. (Think about it: what features of Python did we need in our proof?[4])

DEFINITIONS

There is something counter-intuitive about function `halt`: we can describe what the call `halt(f,i)` is supposed to return, even though the preceding argument shows that there is no way to implement function `halt` in Python (or in any other language, for that matter). This is different from every other Python function you've thought about: usually, you start with a vague, intuitive idea of what you want to accomplish, and you turn it into a working piece of Python code. But in this case, the process does not carry through: we can define the behaviour of function `halt`, but it cannot be implemented.

This shows that there are one-argument functions whose behaviour can be defined clearly, but that cannot be computed by any algorithm. Here is some standard terminology regarding these issues.

Definition: A function $f : A \to B$ is computable if it can be implemented in Python, *i.e.*, if there exists a Python function `f` such that for all $a \in A$, `f(a)` returns the value of $f(a)$.

Functions for which this is not possible (*e.g.*, `halt`) are called non-computable. Note that this definition applies only to well-defined mathematical functions $f : A \to B$ — those for which there is exactly one value $f(a)$ for every $a \in A$. The distinction between "computable" and "non-computable" has nothing to do with the *definition* of the function f. Rather, it distinguishes between functions whose values can be calculated by an algorithm, and those for which this is not possible.

5.3 Reductions

So `halt` is not computable. But it's not the end of the world if this one function cannot be computed, is it?

Unfortunately, non-computable functions are like bugs: when you discover one, you quickly realize that there are many more around... In this section, we'll explain how to use the fact that `halt` is not computable, to show that other functions are also non-computable.

For example, consider the function `initialized(f,v)`, whose value is `True` when variable `v` is guaranteed to be initialized before its first use, whenever Python function `f` is called (no matter what input is passed to `f`) — `initialize(f,v)` is `False` if there is even one input `i` such that the call `f(i)` attempts to use the value of `v` before it has been initialized. Note that there is a subtle, but important, distinction to be made: we don't want `initialized(f,v)` to be `True` when there is just a *possibility* that variable `v` may be used before its initialization in `f` — we want to know that this actually happens during the call `f(i)` for some input `i`. For example, consider the following functions:

```
def f1(x):
    return x + 1
    print y
```

```
def f2(x):
    return x + y + 1
```

While it is the case that `f1` contains a statement that uses variable `y` before it is initialized, this statement can never actually be executed — so `initialized(f1,y)` is `True`, vacuously. On the other hand, variable `y` is actually used before its initialization in `f2` — so `initialized(f2,y)` is `False`.

Claim 5.1: The function `initialized` is non-computable.

Proof:

For a contradiction, assume that `initialized` is computable, *i.e.*, it can be implemented as a Python function.

We want to show that this assumption leads to a contradiction. More specifically in this case, we want to reach the contradiction that `halt` is computable. So, consider the following program.

```
def halt(f,i):
    def initialized(g,v):
        ...code for initialized goes here...

    # Put some code here to scan the code for f and figure out
    # a variable name that doesn't occur in f, and store it in v

    def f_prime(x):
        # Ignore the argument x, call f with the fixed argument i
        # (the one passed in to halt).
        f(i)
        exec("print " + v) # treat string v as an identifier

    return not initialized(f_prime,v)
```

(Although we left out some of the details in this code — the part that is commented out — it *can* be filled in and the result is a valid Python program — assuming that the code for `initialized` can be filled-in, of course.)
If `f(i)` halts, then `f_prime(i)` will execute the statement `print v`, so `initialized(f_prime,v)` returns `False` and `halt(f,i)` returns `True`.
If `f(i)` does not halt, then `f_prime(x)` never executes `print v` (no matter what value `x` has), so `initialized(f_prime,v)` returns `True` and `halt(f,i)` returns `False`.
But there is no Python implementation of function `halt`, as we've already shown!

Hence, there is no Python implementation of function `initialized`.

Let's step through the preceding argument to better understand it.

- We want to prove that `initialized` is non computable (*i.e.*, that there is no Python code to compute it), based on the fact that `halt` is non-computable.

  We decide to use a proof by contradiction, so we begin by assuming that `initialized` *is* computable.

  Our goal is now to derive a contradiction. An obvious candidate is the statement: "`halt` is computable."

- To show that `halt` is computable means to show that there exists an algorithm to compute it. The easiest way to achieve this is to describe an explicit algorithm for `halt`.

  In other words, our goal is to find a way to compute `halt`, given a supposed algorithm for `initialized`.

- The trick to achieving this is contained in the definition of function `f_prime` in the argument: this function is valid Python code and it has been defined with the property that the call `f(i)` halts iff variable `v` is used without being initialized in `f_prime`.

  Moreover, this property does *not* depend on our knowledge of whether or not `f(i)` halts.

- Hence, we know that our implementation of `halt` will return the correct value, under the assumption that `initialized` has been implemented correctly.

- The contradiction follows immediately, which concludes the proof.

Note that the critical step in this argument — the one that requires the most creativity and insight — is to find a way to tie together the fact that "the call `f(i)` halts" with the fact that "variable `v` is always initialized before its use within function `f_prime`." Once we figure out how to achieve this, the structure of the rest of the proof is straightforward.

This kind of argument is called a REDUCTION and can be used to show that many other functions are non-computable, by proving conditional statements of the form:

> *If* f is computable (*i.e.*, there is a Python function `f` that computes f), *then* h is also computable (*i.e.*, we can write a python function `h` that computes h).

Picking h to be a function known to be non-computable (like `halt`) immediately gives us that f is also non-computable, by contrapositive.

5.4 COUNTABILITY

The idea for function `confused` in the proof that `halt` is non-computable is an example of a DIAGONALIZATION argument, first used by the mathematician Georg Cantor to show that the set of real numbers is larger than the set of natural numbers.

Now that, in itself, may seem like a strange statement to make: how can the set of real numbers be "larger" than the set of natural numbers? Don't they both contain infinitely many numbers? What does it even mean to compare the sizes of infinitely large sets?

Let's take things one step at a time and scale back to simpler infinite sets. Consider $\mathbb{N}$ (the set of natural numbers) and $\mathbb{Z}$ (the set of integers). Between the two, which set is LARGER? The answer seems obvious: $\mathbb{Z}$ contains $\mathbb{N}$ as proper subset, so $\mathbb{Z}$ must be larger.

But now think about this: which set has larger SIZE? Now the answer is not so obvious: for finite sets "size" is a natural number, but how do we measure the size of infinite sets? Adding or removing one element from an infinite set does not change its size: it's still infinite. So are all infinite sets the same size? And what does "infinite size" even mean?

Let's think through this more carefully. When we COUNT the elements in a set, what we are really doing is *associating a number with each element*: you can easily imagine yourself pointing to elements one after the other while saying "one, two, three, ..." The easiest way to formalize this idea involves the notions of one-to-one and onto functions.

DEFINITION: Suppose $f : A \to B$ (*i.e.*, f is a function that associates an element $f(a) \in B$ to each element $a \in A$). Then we say that

- f is ONE-TO-ONE if $\forall a_1 \in A, \forall a_2 \in A, f(a_1) = f(a_2) \Rightarrow a_1 = a_2$ (*i.e.*, f gives distinct values to different elements of A);
- f is ONTO if $\forall b \in B, \exists a \in A, f(a) = b$ (*i.e.*, every element in B can be "reached" from at least one element of A).

For example,

- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = \lfloor x/2 \rfloor + 2$ is neither one-to-one ($f(2) = 3 = f(3)$) nor onto (there is no $x \in \mathbb{N}$ such that $f(x) = 1$);
- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = 2x$ is one-to-one ($f(x) = f(y) \Rightarrow x = y$) but not onto (there is no $x \in \mathbb{N}$ such that $f(x) = 1$);
- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = \lfloor x/2 \rfloor$ is not one-to-one ($f(2) = 1 = f(3)$) but it is onto (for all $n \in \mathbb{N}$, $f(2n) = n$);
- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = 10 - x$ if $x \leqslant 10$; $f(x) = x$ otherwise is both one-to-one (the numbers $\{0, \ldots, 10\}$ get mapped to $\{10, \ldots, 0\}$ and $\{11, 12, \ldots\}$ remain unchanged) and onto ($\forall n \in \mathbb{N}, (n \leqslant 10 \Rightarrow f(10 - n) = n) \wedge (n > 10 \Rightarrow f(n) = n)$).

(You can find out more about one-to-one and onto functions in Section 5.2 of Velleman's "How to Prove It," or on Wikipedia.)
Using these concepts, the idea of "counting" can be formalized as follows:

DEFINITION: Set A is COUNTABLE if

1. there is a function $f : A \to \mathbb{N}$ that is one-to-one, or equivalently,
2. there is a function $f : \mathbb{N} \to A$ that is onto.

For example,

CLAIM 5.2: $\mathbb{Z}$ is countable.

Let

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ (1-n)/2 & \text{if } n \text{ is odd.} \end{cases}$$

Then $f : \mathbb{N} \to \mathbb{Z}$.
Next, we show that f is onto.

Assume $x \in \mathbb{Z}$.
Then $x \leqslant 0$ or $x \geqslant 0$.
Case 1: Assume $x \leqslant 0$.
Let $n' = 1 - 2x$.
Then $x \leqslant 0 \Rightarrow -2x \geqslant 0 \Rightarrow n' \geqslant 0$ so $n' \in \mathbb{N}$.
Also, $n' = 2(-x) + 1$ so n' is odd.
Then $f(n') = (1 - n')/2 = (1 - (1 - 2x))/2 = 2x/2 = x$.
Hence $\exists n \in \mathbb{N}, f(n) = x$.
Case 2: Assume $x \geqslant 0$.
Let $n' = 2x$.
Then $x \geqslant 0 \Rightarrow n' \geqslant 0$ so $n' \in \mathbb{N}$.
Also, $n' = 2x$ is even.
Then $f(n') = n'/2 = 2x/2 = x$.
Hence $\exists n \in \mathbb{N}, f(n) = x$.
In all cases, $\exists n \in \mathbb{N}, f(n) = x$.
Since x was arbitrary, $\forall x \in \mathbb{Z}, \exists n \in \mathbb{N}, f(n) = x$, *i.e.*, f is onto. (Note that f is not one-to-one — can you see why?[5])
Hence, there is a function $f : \mathbb{N} \to \mathbb{Z}$ that is onto, *i.e.*, $\mathbb{Z}$ is countable.

Informally, we often show that a set A is countable by giving an argument that it is possible to LIST every element in the set — this corresponds to giving a function $f : \mathbb{N} \to A$ that is onto, though the function may not be written out algebraically. What matters most is to make sure that there is some systematic way to list the elements of A and that the list includes every element of A at least once.

For example, we may argue that $\mathbb{Z}$ is countable by exhibiting the following list (really, more a *list pattern* because of the "..."):

$$\mathbb{Z} : 0, -1, 1, -2, 2, -3, 3, \ldots$$

Implicitly, this defines a function $f : \mathbb{N} \to \mathbb{Z}$ — just think of the list as an enumeration of f's values ($f(0) = 0, f(1) = -1, f(2) = 1, \ldots$) Once we see the pattern, it is obvious that the list includes every possible integer, *i.e.*, the implicit function f is onto. Hence, $\mathbb{Z}$ is countable.

What about finite sets, *e.g.*, $\{a, b, c\}$? Is it countable? Yes, because of condition 1 in the definition of countability: the function $f(a) = 1$, $f(b) = 2$, $f(c) = 3$ is easily proved to be one-to-one.

Assume $x, y \in \{a, b, c\}$ and $f(x) = f(y)$.
Then $f(x) = f(y) = 1$ or $f(x) = f(y) = 2$ or $f(x) = f(y) = 3$.
\# these are the only values in the domain of f
Case 1: Assume $f(x) = f(y) = 1$.
Then $x = y = a$ so $x = y$.
Case 2: Assume $f(x) = f(y) = 2$.
Then $x = y = b$ so $x = y$.
Case 3: Assume $f(x) = f(y) = 3$.
Then $x = y = c$ so $x = y$.
In every case, $x = y$.
Hence $\forall x \in \{a, b, c\}, \forall y \in \{a, b, c\}, f(x) = f(y) \Rightarrow x = y$, *i.e.*, f is one-to-one.

What about other infinite sets? Are they all countable? Consider the set $\mathbb{Q}$ of rational numbers (*i.e.*, numbers of the form p/q for integers p, q with $q \neq 0$). Numerically, we know that this set is much different from $\mathbb{Z}$ or $\mathbb{N}$: the rational numbers are DENSE (there are infinitely many rational numbers between any two other rational numbers). This difference might seem to imply that $\mathbb{Q}$ is not countable, for how could we hope to list all of the rational numbers? However, note that our informal definition of countability does NOT require that we be able to list the elements in any kind of numerical order — in fact, our list for $\mathbb{Z}$ above was not ordered numerically.

In order to show that $\mathbb{Q}$ is countable, we first prove a lemma.

Claim 5.3: $\mathbb{Q}^+$ is countable (where $\mathbb{Q}^+$ is the set of *positive* rational numbers).

We give an informal "list" argument.
Let $f(n)$ be defined implicitly by the following list process: list fractions p/q in increasing order of $p+q$, starting with $p+q=2$, and in increasing order of p within each sub-list with a fixed value of $p+q$. The first few terms of the list are as follows:

- sub-list 0: $1/1$, (fractions p/q where $p+q=2$)
- sub-list 1: $1/2, 2/1$, (fractions p/q where $p+q=3$)
- sub-list 2: $1/3, 2/2, 3/1$, (fractions p/q where $p+q=3$)
- ...

Then $f(n) : \mathbb{N} \to \mathbb{Q}^+$. Also, $f(n)$ is onto: every fraction p/q with $p>0, q>0$ is eventually listed (in position p of sub-list number $p+q-2$). Hence, by definition, $\mathbb{Q}^+$ is countable.

Claim 5.4: $\mathbb{Q}$ is countable.

Note that $\mathbb{Q} = \{0\} \cup \mathbb{Q}^- \cup \mathbb{Q}^+$.
Let $f(n)$ be the function from lemma 5.3 (showing $\mathbb{Q}^+$ is countable).
Then, the following is a complete list of $\mathbb{Q}$, *i.e.*, it defines a function $f' : \mathbb{N} \to \mathbb{Q}$ that is onto:

$$\begin{aligned} &0, f(0), -f(0), f(1), -f(1), f(2), -f(2), \ldots \\ =\ &0, 1/1, -1/1, 1/2, -1/2, 2/1, -2/1, \ldots \end{aligned}$$

By this point, you may start to think that EVERY set is countable, and that the notion is meaningless. However, this turns out not to be the case. To see this, first try to come up with a complete list of $\mathbb{R}$ (the set of real numbers) — to make it easier, you may begin with just $\mathbb{R}^+$, as we did for $\mathbb{Q}$.
If you run into difficulties, remember that

- every real number $r \in \mathbb{R}$ can be written as an infinite decimal expansion of the form $r = m.d_1d_2d_3\ldots$, where $m \in \mathbb{Z}$ and $d_1, d_2, d_3, \ldots \in \{0,1,2,3,4,5,6,7,8,9\}$ and the expansion does NOT end with repeating 9's;
- every infinite decimal expansion of the form $r = m.d_1d_2d_3\ldots$, where $m \in \mathbb{Z}$ and $d_1, d_2, d_3, \ldots \in \{0,1,2,3,4,5,6,7,8,9\}$ and the expansion does NOT end with repeating 9's, defines a *unique* real number $r \in \mathbb{R}$.

The provision that the decimal expansion does not end with repeating 9's is a consequence of the positional notation system, and the fact that $0.\overline{9}\ldots = 1.\overline{0}\ldots$ Hence, every decimal expansion that ends with infinitely many 9's is equivalent to a decimal expansion that ends with infinitely many 0's instead — but all other decimal expansions are different from each other.

5.5 Diagonalization

Cantor's proof

Try as you might, you will not be able to provide a complete list of $\mathbb{R}$. Georg Cantor showed that this was impossible through the following proof, called a DIAGONALIZATION ARGUMENT.

Claim 5.5: $\mathbb{R}$ is uncountable.

PROOF:

For a contradiction, assume that $\mathbb{R}$ is countable.

Then there is a function $f : \mathbb{N} \to \mathbb{R}$ that is onto. # by definition of "countable"

We can visualize function f as follows: for every natural number $n \in \mathbb{N}$, $f(n)$ is a real number, represented as an infinite decimal expansion that does *not* end with repeating 9's:

$$\begin{aligned}
f(0) &= i_0.d_{0,0}\,d_{0,1}\,d_{0,2}\cdots d_{0,n}\cdots \\
f(1) &= i_1.d_{1,0}\,d_{1,1}\,d_{1,2}\cdots d_{1,n}\cdots \\
f(2) &= i_2.d_{2,0}\,d_{2,1}\,d_{2,2}\cdots d_{2,n}\cdots \\
\vdots \quad &= \qquad\qquad \vdots \\
f(n) &= i_n.d_{n,0}d_{n,1}d_{n,2}\cdots d_{n,n}\cdots \\
\vdots \quad &= \qquad\qquad \vdots
\end{aligned}$$

where $i_0, i_1, \ldots \in \mathbb{Z}$ are the integer parts of the real numbers $f(0), f(1), \ldots$, and $\forall i \in \mathbb{N}, \forall j \in \mathbb{N}$, $d_{i,j} \in \{0,1,2,3,4,5,6,7,8,9\}$.

Now, let $r = 0.d_0d_1d_2\cdots d_n\cdots$, where $\forall i \in \mathbb{N}$,

$$d_i = \begin{cases} 1 & \text{if } d_{i,i} = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then $r \in \mathbb{R}$. # r is an infinite decimal that does not end with repeating 9's

Then $\exists k \in \mathbb{N}, f(k) = r$. # f is onto

Since $f(k) = i_k.d_{k,0}d_{k,1}d_{k,2}\cdots d_{k,n}\cdots$ and $r = 0.d_0d_1d_2\cdots d_n\cdots$, this implies $i_k = 0$ and $\forall n \in \mathbb{N}$,

$$d_{k,n} = d_n = \begin{cases} 1 & \text{if } d_{n,n} = 0, \\ 0 & \text{otherwise.} \end{cases}$$

But then,

$$d_{k,k} = d_k = \begin{cases} 1 & \text{if } d_{k,k} = 0, \\ 0 & \text{otherwise,} \end{cases}$$

i.e., $d_{k,k} = 0 \Leftrightarrow d_{k,k} = 1$, a contradiction!

Then, by contradiction, $\mathbb{R}$ is not countable.

Take the time to study this proof a little...

You should notice the following key elements:

- The construction of a real number r such that $\forall n \in \mathbb{N}, f(n) \neq r$.
- The fact that this construction is carried out for an arbitrary function $f : \mathbb{N} \to \mathbb{R}$.

You should be able to use these ideas to write a direct proof that there does not exist any function $f : \mathbb{N} \to \mathbb{R}$ that is onto — thus proving directly that $\mathbb{R}$ is uncountable.

DIAGONALIZATION AND COMPUTABILITY

The construction of real number r in the last proof is done using diagonalization. The reason for this name becomes obvious if we visualize how r is constructed: each digit d_i of r is defined to be different from at least one digit $d_{i,i}$ in the decimal expansion of the real number $f(i)$.

$$
\begin{array}{rcl}
r = & 0. & \boxed{d_0}\ \boxed{d_1}\ \boxed{d_2} \cdots \boxed{d_n} \cdots \\
f(0) = & i_0. & \boxed{d_{0,0}}\ d_{0,1}\ d_{0,2} \cdots d_{0,n} \cdots \\
f(1) = & i_1. & d_{1,0}\ \boxed{d_{1,1}}\ d_{1,2} \cdots d_{1,n} \cdots \\
f(2) = & i_2. & d_{2,0}\ d_{2,1}\ \boxed{d_{2,2}} \cdots d_{2,n} \cdots \\
\vdots & = & \qquad\qquad\ \vdots \\
f(n) = & i_n. & d_{n,0}\ d_{n,1}\ d_{n,2} \cdots \boxed{d_{n,n}} \cdots \\
\vdots & = & \qquad\qquad\ \vdots
\end{array}
$$

Because there are infinitely many digits in the decimal expansion of r (one for each natural number $n \in \mathbb{N}$), r is different from the real number $f(n)$ for every $n \in \mathbb{N}$.

Now, remember that we started this discussion in the context of the proof that function `halt` cannot be implemented in Python. Back then, I mentioned that this proof was an example of diagonalization — I will now explain why.

First, notice that `halt` takes two arguments: a one-argument Python function `f` and an input `i` for `f`. Clearly, there are infinitely many one-argument Python functions `f`, but what kind of infinity: countable or uncountable? To answer this question, we need a specific point of view: remember that every Python function can be written down as a text file, *i.e.*, a finite sequence of characters, from a fixed set of characters. To be specific, let's assume the character set is UTF-8, which contains 256 different characters — the exact details of the encoding used for the set of characters don't matter: what matters is that the set of possible characters is fixed, *i.e.*, the *same* finite set of characters can be used to write down every possible Python function. So, every Python function can be written down as a sequence of characters in UTF-8, and every such sequence of characters is really just an integer in disguise: just treat each character as a "digit" in base-256 notation.

If you think about it, what we've just done in the argument above is to define a function

$$g : \text{one-argument Python functions} \to \mathbb{N}$$

that is one-to-one: different Python functions get assigned to different numbers, because their source code must differ by at least one character. So by definition, the set of one-argument Python functions is countable — actually, the argument is more general than this: it shows that the set of *all* Python programs is countable.

By the definition of countability, this means that there is some other function

$$g' : \mathbb{N} \to \text{one-argument Python functions}$$

that is onto, *i.e.*, it is possible to *list* every one-argument Python function: $g'(0), g'(1), g'(2), \ldots$ — to make the notation easier to understand, we'll use subscripts $f_0 = g'(0), f_1 = g'(1), f_2 = g'(2), \ldots$ in the rest of the argument.

Keep in mind that each f_n is actually just a *string*: the source code for some one-argument Python function. So it makes sense to consider the function call $f_i(f_j)$ for $i, j \in \mathbb{N}$: we are simply calling the Python function whose source code is given by f_i with the string f_j as argument.

Now, consider the following table of behaviours, where we list one-argument Python functions down the left side and inputs for those functions across the top — our list only contains specific kinds of inputs (those that are listings for one-argument Python functions), and it is therefore not a complete list of all possible inputs. The entry at row f_i and column f_j states whether the call $f_i(f_j)$ halts or not (the values shown below are just an example).

	f_0	f_1	f_2 $\cdots$	f_n $\cdots$
f_0	halts	halts	loops $\cdots$	halts $\cdots$
f_1	loops	loops	loops $\cdots$	halts $\cdots$
f_2	loops	halts	halts $\cdots$	loops $\cdots$
$\vdots$			$\vdots$	
f_n	halts	halts	halts $\cdots$	halts $\cdots$
$\vdots$			$\vdots$	

(Note that, when we assume that `halt` can be coded up in Python, this implies that every entry in this table can be computed by `halt`.) The function `confused` is created by a simple process of diagonalization over the table: just use `halt` to figure out the behaviour of $f_i(f_i)$ and make `confused` do the opposite. If `halt` exists, then so does `confused`, but `confused` cannot be any of the elements of the list $f_0, f_1, f_2, \ldots$ — by construction, its behaviour differs from each one of these functions. This is how we get a contradiction.

Note that this argument also shows that *every* list of one-argument functions is incomplete. If you look at if again more carefully, you should see that the argument can be interpreted as a proof that the set of one-argument functions (meaning functions' behaviours) is uncountable. Since we've already argued that the set of Python programs is countable, this means there are uncountably many functions that cannot be implemented in Python!

We've barely scratched the surface of this fascinating topic. If you find it interesting — and who wouldn't! — then you can find out much more in the courses CSC 463 H and CSC 438 H.

CHAPTER 5 NOTES

[1]Yes, it would. And it turns out to be possible to give just such an argument, as you'll see.

[2]Yes, Python does allow such "nested" function definitions: it simply makes `halt` defined locally within `confused`, just like for variables.

[3]Notice this is just an application of Excluded Middle.

[4]Functions, conditionals, and loops. We used "nested functions," but that is not strictly necessary.

[5]$f(0) = 0/2 = 0 = (1-1)/2 = f(1)$.