David Liu

UTM edits by Daniel Zingaro

Introduction to the Theory of Computation

Lecture Notes and Exercises for CSC236

Department of Computer Science
University of Toronto

Contents

Introduction

There is a common misconception held by our students, the students of other disciplines, and the public at large about the nature of computer science. So before we begin our study this semester, let us clear up exactly what we will be learning: computer science is *not* the study of programming, any more than chemistry is the study of test tubes or math the study of calculators. To be sure, programming ability is a vital tool in any computer scientist's repertoire, but it is still a tool in service to a higher goal.

Computer science is the study of *problem-solving*. Unlike other disciplines, where researchers use their skill, experience, and luck to solve problems at the frontier of human knowledge, computer science asks: What is problem-solving? How are problems solved? Why are some problems easier to solve than others? How can we measure the quality of a problem's solution?

Even the many who go into industry confront these questions on a daily basis in their work!

It should come as no surprise that the field of computer science predates the invention of computers, since humans have been solving problems for millennia. Our English word *algorithm*, a sequence of steps taken to solve a problem, is named after the Persian mathematician Muhammad ibn Musa al-Khwarizmi, whose mathematics texts were compendia of mathematics computational procedures. In 1936, Alan Turing, one of the fathers of modern computer science, developed the *Turing Machine*, a theoretical model of computation which is widely believed to be just as powerful as all programming languages in existence today. In one of the earliest and most fundamental results in computer science, Turing proved that there are some problems that cannot be solved by any computer that has ever or will ever be built – before computers had been invented at all!

The word *algebra* is derived from the word *al-jabr*, appearing in the title of one of his books, describing the operation of subtracting a number from both sides of an equation.

A little earlier, Alonzo Church (who would later supervise Turing during the latter's graduate studies) developed the *lambda calculus*, an alternative model of computation that forms the philosophical basis for functional programming languages like Scheme, Haskell, and ML.

But Why Do I Care?

A programmer's value lies not in her ability to write code, but to understand problems and design solutions – a much harder task. Beginning programmers often write code by trial and error ("Does this compile? What if I add this line?"), which indicates not a lack of programming experience, but a lack of *design* experience. When presented with a problem, many students often jump straight to the computer, even if they have no idea what they are going to write! And when the code is complete, they are at a loss when asked the two fundamental questions: Why is your code correct, and is it a good solution?

"My code is correct because it passed all of the tests" is reasonable but unsatisfying. What I really want to know is *how* your code works.

In this course, you will learn the skills necessary to answer both of these questions, improving both your ability to reason about the code you write and your ability to communicate your thinking with others. These skills will help you design cleaner and more efficient programs, and clearly document and present your code. Of course, like all skills, you will practice and refine these throughout your university education and in your careers.

Overview of this Course

The first section of the course introduces the powerful proof technique of *induction*. We will see how inductive arguments can be used in many different mathematical settings; you will master the structure and style of inductive proofs, so that later in the course you will not even blink when asked to read or write a "proof by induction."

From induction, we turn our attention to the runtime analysis of recursive programs. You have done this already for non-recursive programs, but did not have the tools necessary to handle recursion. We will see that (mathematical) induction and (programming) recursion are two sides of the same coin, so we use induction to make analysing recursive programs easy as cake. After these lessons, you will always be able to evaluate your recursive code based on its runtime, a very important consideration!

Some might even say, chocolate cake.

We next turn our attention to the *correctness* of both recursive and non-recursive programs. You already have some intuition about why your programs are correct; we will teach you how to formalize this intuition into mathematically rigorous arguments, so that you may reason about the code you write and determine errors *without* the use of testing.

This is not to say tests are unnecessary! The methods we'll teach you in this course are quite tricky for larger software systems. However, a more mature understanding of your own code certainly facilitates finding and debugging errors.

Finally, we will turn our attention to the simplest model of computation, the finite automaton. This serves as both an introduction to more complex computational models like Turing Machines, and also formal language theory through the intimate connection between finite automata and regular languages. Regular languages and automata have many other applications in computer science, from text-based pattern matching to modelling biological processes.

Prerequisite Knowledge

CSC236 is mainly a theoretical course, the successor to MAT102. This is fundamentally a computer science course, though, so while mathematics will play an important role in our thinking, we will mainly draw motivation from computer science concepts. Also, you will be expected to both read and write Python code at the level of CSC148. Here are some brief reminders about things you need to know – and if you find that you don't remember something, review early!

Concepts from MAT102

In MAT102, you learned how to write proofs. This is the main object of interest in CSC236, so you should be comfortable with this style of writing. However, one key difference is that we will not expect (nor award marks for) a particular proof structure – indentation is no longer required, and your proofs can be mixtures of mathematics, English paragraphs, pseudocode, and diagrams! Of course, we will still greatly value clear, well-justified arguments, *especially* since the content will be more complex.

So a technically correct solution that is extremely difficult to understand will not receive full marks. Conversely, an incomplete solution which explains clearly partial results (and possibly even what is left to do to complete the solution) will be marked more generously.

Concepts from CSC148

Recursion, recursion, recursion. If you liked using recursion CSC148, you're in luck: induction, the central proof structure in this course, is the abstract thinking behind designing recursive functions. And if you didn't like recursion or found it confusing, don't worry! This course will give you a great opportunity to develop a better feel for recursive functions in general, and even give you programming opportunities to get practical experience.

This is not to say you should forget everything you have done with iterative programs; loops will be present in our code throughout this

course, and will be the central object of study for a week or two when we discuss program correctness. In particular, you should be very comfortable with the central *design pattern* of first-year python: computing on a list by processing its elements one at a time using a `for` or `while` loop.

A *design pattern* is a common coding template which can be used to solve a variety of different problems. "Looping through a list" is arguably the simplest one.

You should also be comfortable with terminology associated with *trees*, which will come up occasionally throughout the course when we discuss induction proofs.

You will also have to remember the fundamentals of Big-O algorithm analysis, and how to determine tight asymptotic bounds for common functions.

Finally, the last part of the course deals with regular languages; you should be familiar with the terminology associated with *strings*, including length, reversal, concatenation, and the empty string.

Induction

What is the sum of the numbers from 0 to n? This is a well-known identity you've probably seen before:

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

A "proof" of this is attributed to Gauss:

$$\begin{aligned} 1+2+3+\cdots+n-1+n &= (1+n)+(2+n-1)+(3+n-2)+\cdots \\ &= (n+1)+(n+1)+(n+1)+\cdots \\ &= \frac{n}{2}(n+1) \qquad \text{(since there are } \frac{n}{2} \text{ pairs)} \end{aligned}$$

This isn't exactly a formal proof – what if n is odd? – and although it could be made into one, this proof is based on a mathematical "trick" that doesn't work for, say, $\sum_{i=0}^{n} i^2$. And while mathematical tricks are often useful, they're hard to come up with in the first place! Induction gives us a different way to tackle this problem that is astonishingly straightforward.

We ignore the 0 in the summation, since this doesn't change the sum.

A *predicate* is a parametrized logical statement. Another way to view a predicate is as a function that takes in one or more arguments, and outputs either **True** or **False**. Some examples of predicates are:

$EV(n)$: n is even

$GR(x, y)$: $x > y$

$FROSH(a)$: a is a first-year university student

Every predicate has a *domain*, the set of its possible input values. For example, the above predicates could have domains $\mathbb{N}$, $\mathbb{R}$, and "the set of all UofT students," respectively. Predicates give us a precise way of formulating English problems; the predicate that is relevant to our example is

We will always use the convention that $0 \in \mathbb{N}$ unless otherwise specified.

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

A common mistake: defining the predicate to be something like $P(n) : \frac{n(n+1)}{2}$. Such an expression is wrong and misleading because *it isn't a True/False value*, and so fails to capture precisely what we want to prove.

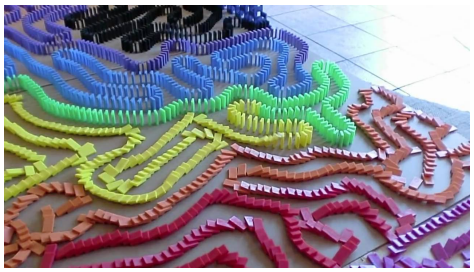
You might be thinking right now: "Okay, now we're going to prove that $P(n)$ is true." But this is wrong, because we haven't yet defined n! So in fact we want to prove that $P(n)$ is true *for all* natural numbers n, or written symbolically, $\forall n \in \mathbb{N}, P(n)$. Here is how a formal proof might go if we were not using mathematical induction:

$$\text{Proof of } \forall n \in \mathbb{N}, \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Let $n \in \mathbb{N}$.
    `# Want to prove that` $P(n)$ `is true.`
    Case 1: **Assume** n is even.
        `# Gauss' trick`
        ⋮
        Then $P(n)$ is true.
    Case 2: **Assume** n is odd.
        `# Gauss' trick, with a twist?`
        ⋮
        Then $P(n)$ is true.
    Then in all cases, $P(n)$ is true.
Then $\forall n \in \mathbb{N}, P(n)$. ■

Instead, we're going to see how induction gives us a different, easier way of proving the same thing.

The Induction Idea



Suppose we want to create a viral Youtube video featuring "The World's Longest Domino Chain!!! (like plz)".

Of course, a static image like the one featured on the right is no good for video; instead, once we have set it up we plan on recording all of the dominoes falling in one continuous, epic take. It took a lot of effort to set up the chain, so we would like to make sure that it will work; that is, that once we tip over the first domino, all the rest will fall. Of course, with dominoes the idea is rather straightforward, since we have arranged the dominoes precisely enough that any one falling will trigger the next one to fall. We can express this thinking a bit more formally:

(1) The first domino will fall (when we push it).
(2) For each domino, *if* it falls, *then* the next one in the chain will fall (because each domino is close enough to the next one).

From these two thoughts, we can conclude that

(3) Every domino in the chain will fall.

We can apply the same reasoning to the set of natural numbers. Instead of "every domino in the chain will fall," suppose we want to prove that "for all $n \in \mathbb{N}$, $P(n)$ is true", where $P(n)$ is some predicate. The analogues of the above statements in the context of natural numbers are

(1) $P(0)$ is true (0 is the "first" natural number)
(2) $\forall k \in \mathbb{N}, P(k) \Rightarrow P(k+1)$
(3) $\forall n \in \mathbb{N}, P(n)$ is true

The "is true" is redundant, but we will often include these words for clarity.

Putting these together yields the *Principle of Simple Induction* (also known as Mathematical Induction):

simple/mathematical induction

$$\big(P(0) \wedge \forall k \in \mathbb{N}, P(k) \Rightarrow P(k+1)\big) \Rightarrow \forall n \in \mathbb{N}, P(n)$$

A different, slightly more mathematical intuition for what induction says is that "$P(0)$ is true, and $P(1)$ is true *because* $P(0)$ is true, and $P(2)$ is true *because* $P(1)$ is true, and $P(3)$ is true because…" However, it turns out that a more rigorous *proof* of simple induction doesn't exist from the basic arithmetic properties of the natural numbers alone. Therefore mathematicians accept the principle of induction as an *axiom*, a statement as fundamentally true as $1 + 1 = 2$.

It certainly makes sense intuitively, and turns out to be equivalent to another fundamental math fact called the *Well-Ordering Principle*.

This gives us a new way of proving a statement is true for all natural numbers: instead of proving $P(n)$ for an arbitrary n, just prove $P(0)$, and then prove the *link* $P(k) \Rightarrow P(k+1)$ for an arbitrary k. The former step is called the *base case*, while the latter is called the *induction step*. We'll see exactly how such a proof goes by illustrating it with the opening example.

Example. Prove that for every natural number n, $\displaystyle\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

The first few induction examples in this chapter have a great deal of structure; this is only to help you learn the necessary ingredients of induction proofs. We will not be marking for a particular structure in this course, but you will probably find it helpful to use our keywords to organize your proofs.

Proof. First, we define the *predicate* associated with this question. This lets us determine exactly what it is we're going to use in the induction proof.

Step 1 (Define the Predicate) $\displaystyle P(n) : \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

It's easy to miss this step, but without it, often you'll have trouble deciding precisely what to write in your proofs.

Step 2 (Base Case): $n = 0$. We would like to prove that $P(0)$ is true. Recall the meaning of P:

$$P(0): \quad \sum_{i=0}^{0} i = \frac{0(0+1)}{2}.$$

For induction proofs, the base case usually a very straightforward proof. In fact, if you find yourself stuck on the base case, then it is likely that you've misunderstood the question and/or are trying to prove the wrong predicate.

This statement is trivially true, because both sides of the equation are equal to 0.

Step 3 (Induction Step): the goal is to prove that $\forall k \in \mathbb{N}, P(k) \Rightarrow P(k+1)$. Let $k \in \mathbb{N}$ be some arbitrary natural number, and assume $P(k)$ is true. This antecedent assumption has a special name: the **Induction Hypothesis**. Explicitly, we assume that

$$\sum_{i=0}^{k} i = \frac{k(k+1)}{2}.$$

Now, we want to prove that $P(k+1)$ is true, i.e., that $\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$. This can be done with a simple calculation:

$$\begin{aligned}
\sum_{i=0}^{k+1} i &= \left(\sum_{i=0}^{k} i\right) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1) && \textbf{(By Induction Hypothesis)} \\
&= (k+1)\left(\frac{k}{2} + 1\right) \\
&= \frac{(k+1)(k+2)}{2}
\end{aligned}$$

We break up the sum by removing the last element.

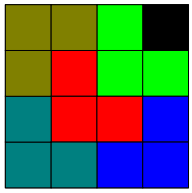
The one structural requirement we do have for this course is that you must always state exactly where you use the induction hypothesis. We expect to see the words "by the induction hypothesis" at least once in each of your proofs.

Therefore $P(k+1)$ holds. This completes the proof of the induction step: $\forall k \in \mathbb{N}, P(k) \Rightarrow P(k+1)$.

Finally, by the Principle of Simple Induction, we can conclude that $\forall n \in \mathbb{N}, P(n)$. ■

In our next example, we look at a geometric problem – notice how our proof will use no algebra at all, but instead constructs an argument from English statements and diagrams. This example is also interesting because it shows how to apply simple induction starting at a number other than 0.

Example. A *triomino* is a three-square L-shaped figure. To the right, we show a 4-by-4 chessboard with one corner missing that has been *tiled* with triominoes.



Prove that for all $n \geq 1$, any 2^n-by-2^n chessboard with one corner missing can be tiled with triominoes.

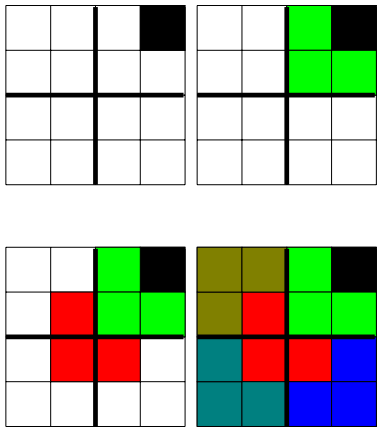
Proof. **Predicate:** $P(n)$: Any 2^n-by-2^n chessboard with one corner missing can be tiled with triominoes.

Base Case: This is slightly different, because we only want to prove the claim for $n \geq 1$ (and ignore $n = 0$). Therefore our base case is $n = 1$, i.e., this is the "start" of our induction chain. When $n = 1$, we consider a 2-by-2 chessboard with one corner missing. But such a chessboard is exactly the same shape as a triomino, so of course it can be tiled by triominoes!

Again, a rather trivial base case. Keep in mind that even though it was simple, the proof would have been incomplete without it!

Induction Step: Let $k \geq 1$ and suppose that $P(k)$ holds; that is, that every 2^k-by-2^k chessboard with one corner missing can be tiled by triominoes. (This is the *Induction Hypothesis*.) The *goal* is to show that any 2^{k+1}-by-2^{k+1} chessboard with one corner missing can be tiled by triominoes.

Consider an arbitrary 2^{k+1}-by-2^{k+1} chessboard with one corner missing. Divide it into quarters, each quarter a 2^k-by-2^k chessboard.



Exactly one of these has one corner missing; **by the Induction Hypothesis**, this quarter can be tiled by triominoes. Next, place a single triomino in the middle that covers one corner in each of the three remaining quarters.

Each of these quarters now has one corner covered, and **by the I.H.** again, they can each be tiled by triominoes. This completes the tiling of the 2^{k+1}-by-2^{k+1} chessboard. ■

Note that in this proof, we used the induction hypothesis twice! (Or technically, 4 times, one for each 2^k-by-2^k quarter.)

Before moving on, here is some intuition behind what we did in the previous two examples. Given a problem of a 2^n-by-2^n chessboard, we repeatedly broke it up into smaller and smaller parts, until we reached the 2-by-2 size, which we could tile using just a single triomino. This idea of breaking down the problem into smaller ones "again and again" was a clear sign that a formal proof by induction was the way to go. Be on the lookout for phrases like "repeat over and over" in your own thinking to signal that you should be using induction. In the opening example, we used an even more specific approach: in the induction step, we took the sum of size $k+1$ and reduced it to a sum of size k, and evaluated that using the induction hypothesis. The cornerstone of simple induction is this link between problem instances of size k and size $k+1$, and this ability to break down a problem into something *exactly one size smaller*.

In your programming, this is the same sign that points to using recursive solutions as the easiest approach.

Example. Consider the sequence of natural numbers satisfying the following properties: $a_0 = 1$, and for all $n \geq 1$, $a_n = 2a_{n-1} + 1$. Prove that for all $n \in \mathbb{N}$, $a_n = 2^{n+1} - 1$.

We will see in the next chapter one way of discovering this expression for a_n.

Proof. The **predicate** we will prove is

$$P(n) : a_n = 2^{n+1} - 1.$$

The **base case** is $n = 0$. By the definition of the sequence, $a_0 = 1$, and $2^{0+1} - 1 = 2 - 1 = 1$, so $P(0)$ holds.

For the **induction step**, let $k \in \mathbb{N}$ and suppose $a_k = 2^{k+1} - 1$. Our goal is to prove that $P(k+1)$ holds. By the recursive property of the sequence,

$$\begin{aligned} a_{k+1} &= 2a_k + 1 \\ &= 2(2^{k+1} - 1) + 1 && \text{(by the I.H.)} \\ &= 2^{k+2} - 2 + 1 \\ &= 2^{k+2} - 1 \end{aligned}$$

■

When Simple Induction Isn't Enough

By this point, you have done several examples using simple induction. Recall that the intuition behind this proof technique is to reduce problems of size $k+1$ to problems of size k (where "size" might mean the value of a number, or the size of a set, or the length of a string, etc.). However, for many problems there is no natural way to reduce problem sizes just by 1. Consider, for example, the following problem:

> Prove that every natural number greater than 1 has a *prime factorization*, i.e., can be written as a product of primes.

Every prime can be written as a product of just one number: itself!

How would you go about proving the induction step, using the method we've used so far? That is, how would you prove $P(k) \Rightarrow P(k+1)$? This is a very hard question to answer, because even the prime factorizations of consecutive numbers can be completely different!

E.g., $210 = 2 \cdot 3 \cdot 5 \cdot 7$, but 211 is prime.

But if I asked you to solve this question by "breaking the problem down," you would come up with the idea that if $k+1$ is *not* prime, then we can write $k+1 = a \cdot b$, where $a, b < k+1$, and we can "do this recursively" until we're left with a product of primes. Since we always identify *recursion* with *induction*, this hints at a more general form of induction that we can use to prove this statement.

Complete Induction

Recall the intuitive "chain of reasoning" that we do with simple induction: first we prove $P(0)$, and then use $P(0)$ to prove $P(1)$, then use $P(1)$ to prove $P(2)$, etc. So when we get to $k+1$, we try to prove $P(k+1)$ using $P(k)$, but we have already gone through proving $P(0)$, $P(1)$, ..., and $P(k-1)$, in addition to $P(k)$! In some sense, in Simple Induction we're throwing away all of our previous work except for $P(k)$. In *Complete Induction*, we keep this work and use it in our proof of the induction step. Here is the formal statement of **The Principle of Complete Induction**:

complete induction

$$\Big(P(0) \wedge \forall k, \big(P(0) \wedge P(1) \wedge \cdots \wedge P(k)\big) \Rightarrow P(k+1)\Big) \Rightarrow \forall n, P(n)$$

The only difference between Complete and Simple Induction is in the antecedent of the inductive part: instead of assuming just $P(k)$, we now assume all of $P(0)$, $P(1)$, ..., $P(k)$. Since these are *assumptions* we get to make in our proofs, Complete Induction proofs are often more flexible than Simple Induction proofs — intuitively, because we have "more to work with."

Somewhat surprisingly, everything we can prove with Complete Induction we can also prove with Simple Induction, and vice versa. So these proof techniques are equally powerful.

Azadeh Farzan gave a great analogy for the two types of induction. Simple Induction is like a person climbing stairs step by step; Complete Induction is like a robot with multiple giant legs, capable of jumping from any lower step to a higher step.

Let's illustrate this (slightly different) technique by proving the earlier claim about prime factorizations.

Example. Prove that every natural number greater than 1 has a prime factorization.

Proof. **Predicate:** $P(n)$: "There are primes $p_1, p_2, \ldots, p_m$ (for some $m \geq 1$) such that $n = p_1 p_2 \cdots p_m$." We will show that $\forall n \geq 2, P(n)$.

Base Case: $n = 2$. Since 2 is prime, we can let $p_1 = 2$ and say that $n = p_1$, so $P(2)$ holds.

Induction Step: Here is the only structural difference for Complete Induction proofs. We let $k \geq 2$, and our *induction hypothesis* is now to assume that *for all* $2 \leq i \leq k$, $P(i)$ holds. (That is, we're assuming $P(2), P(3), P(4), \ldots, P(k)$ are all true.) The goal is still the same: prove that $P(k+1)$ is true.

There are two cases. In the first case, assume $k+1$ is prime. Then of course $k+1$ can be written as a product of primes, so $P(k+1)$ is true.

The product contains a single number, $k+1$.

In the second case, $k+1$ is composite. But then by the definition of compositeness, there exist $a, b \in \mathbb{N}$ such that $k+1 = ab$ and $2 \leq a, b \leq$

k; that is, $k+1$ has factors other than 1 and itself. This is the intuition from earlier. And here is the "recursive thinking": **by the induction hypothesis**, $P(a)$ and $P(b)$ hold. Therefore we can write

We can only use the induction hypothesis *because* a and b are at least 2 and less than $k+1$.

$$a = q_1 \cdots q_{l_1} \quad \text{and} \quad b = r_1 \cdots r_{l_2},$$

where each of the q's and r's is prime. But then

$$k+1 = ab = q_1 \cdots q_{l_1} r_1 \cdots r_{l_2},$$

and this is the prime factorization of $k+1$. So $P(k+1)$ holds. ■

Note that we used inductive thinking to break down the problem; but unlike Simple Induction where the size of the subproblem is one less than the current problem size, we didn't know much about the sizes of the resulting problems (only that they were smaller than the original problem). Complete Induction allows us to handle this sort of structure.

Example. The *Fibonacci sequence* is a sequence of natural numbers defined recursively as $f_1 = f_2 = 1$, and for all $n \geq 3$, $f_n = f_{n-1} + f_{n-2}$. Prove that for all $n \geq 1$,

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

Proof. Note that we really need complete induction here (and not just simple induction) because f_n is defined in terms of both f_{n-1} and f_{n-2}, and not just f_{n-1} only.

The **predicate** we will prove is $P(n) : f_n = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}$. We require two **base cases**: one for $n = 1$, and one for $n = 2$. These can be checked by simple calculations:

$$\begin{aligned}
\frac{(\frac{1+\sqrt{5}}{2})^1 - (\frac{1-\sqrt{5}}{2})^1}{\sqrt{5}} &= \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} \\
&= \frac{\sqrt{5}}{\sqrt{5}} \\
&= 1 = f_1 \\
\frac{(\frac{1+\sqrt{5}}{2})^2 - (\frac{1-\sqrt{5}}{2})^2}{\sqrt{5}} &= \frac{\frac{6+2\sqrt{5}}{4} - \frac{6-2\sqrt{5}}{4}}{\sqrt{5}} \\
&= \frac{\sqrt{5}}{\sqrt{5}} \\
&= 1 = f_2
\end{aligned}$$

For the induction step, let $k \geq 2$ and assume $P(1), P(2), \ldots, P(k)$ hold. Consider f_{k+1}. By the recursive definition, we have

$$\begin{aligned}
f_{k+1} &= f_k + f_{k-1} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^k - (\frac{1-\sqrt{5}}{2})^k}{\sqrt{5}} + \frac{(\frac{1+\sqrt{5}}{2})^{k-1} - (\frac{1-\sqrt{5}}{2})^{k-1}}{\sqrt{5}} && \text{(by I.H.)} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^k + (\frac{1+\sqrt{5}}{2})^{k-1}}{\sqrt{5}} - \frac{(\frac{1-\sqrt{5}}{2})^k + (\frac{1-\sqrt{5}}{2})^{k-1}}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{k-1}(\frac{1+\sqrt{5}}{2} + 1)}{\sqrt{5}} - \frac{(\frac{1-\sqrt{5}}{2})^{k-1}(\frac{1-\sqrt{5}}{2} + 1)}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{k-1} \cdot \frac{6+2\sqrt{5}}{4}}{\sqrt{5}} - \frac{(\frac{1-\sqrt{5}}{2})^{k-1} \cdot \frac{6-2\sqrt{5}}{4}}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{k-1}(\frac{1+\sqrt{5}}{2})^2}{\sqrt{5}} - \frac{(\frac{1-\sqrt{5}}{2})^{k-1}(\frac{1-\sqrt{5}}{2})^2}{\sqrt{5}} \\
&= \frac{(\frac{1+\sqrt{5}}{2})^{k+1}}{\sqrt{5}} - \frac{(\frac{1-\sqrt{5}}{2})^{k+1}}{\sqrt{5}}
\end{aligned}$$

■

Beyond Numbers

So far, our proofs have all been centred on natural numbers. Even in situations where we have proved statements about other objects — like sets and chessboards — our proofs have always required associating these objects with natural numbers. Consider the following problem:

> Prove that any non-empty binary tree has exactly one more node than edge.

We could use either simple or complete induction on this problem by associating every tree with a natural number (height and number of nodes are two of the most common). But this is not the most "natural" way of approaching this problem (though it's perfectly valid!) because binary trees already have a lot of nice recursive structure that we should be able to use directly, *without* shoehorning in natural numbers. What we want is a way of proving statements about objects *other than* numbers. Thus we move away from $\mathbb{N}$ (the set of natural numbers), to more general sets (such as the set of all non-empty binary trees).

Recursive Definitions of Sets

You are already familiar with many descriptions of sets: $\{2, \pi, \sqrt{10}\}$, $\{x \in \mathbb{R} \mid x \geq 4\}$, and "the set of all non-empty binary trees" are all perfectly valid descriptions of sets. Unfortunately, these set descriptions don't lend themselves very well to induction, because induction is recursion and it isn't clear how to apply recursive thinking to any of these descriptions. However, for some objects – like binary trees – it is relatively straightforward to define them recursively. Here's a warm-up.

Example. Suppose we want to construct a recursive definition of $\mathbb{N}$. Here is one way. Define $\mathbb{N}$ to be the (smallest) set such that:

The "smallest" means that nothing else is in $\mathbb{N}$. This is an important point to make; for example, the set of integers $\mathbb{Z}$ also satisfies the given properties, but includes more than $\mathbb{N}$. In the recursive definitions below, we omit "smallest" but it is always implicitly there.

- $0 \in \mathbb{N}$
- If $k \in \mathbb{N}$, then $k + 1 \in \mathbb{N}$

Notice how similar this definition looks to the Principle of Simple Induction! This isn't a coincidence: induction fundamentally makes use of this recursive structure of $\mathbb{N}$. We'll refer to the first rule as the *base* of the definition, and the second as the *recursive rule*. In general, a recursive definition can have **multiple base and recursive rules!**

Example. Construct a recursive definition of "the set of all non-empty binary trees."

Intuitively, the base rule(s) always capture the smallest or simplest elements of a set. Certainly the smallest non-empty binary tree is a *single node*.

What about larger trees? This is where "breaking down" problems into smaller subproblems makes the most sense. You should know from CSC148 that we really store binary trees in a recursive manner: every tree has a root node and links to the roots of the left and right subtrees (the suggestive word here is "*sub*tree.") One slight subtlety is that one or both of these subtrees could be empty. Here is a formal recursive definition (before you read it, try coming up with one yourself!):

- A single node is a non-empty binary tree.
- If T_1, T_2 are two non-empty binary trees, then the tree with a new root r connected to the roots of T_1 and T_2 is a non-empty binary tree.

- If T_1 is a non-empty binary tree, then the tree with a new root r connected to the root of T_1 to the left or to the right is a non-empty binary tree.

Notice that this definition has two recursive rules, not one!

Structural Induction

Now, we mimic the format of our induction proofs, but with the recursive definition of non-empty binary trees rather than natural numbers. The similarity of form is why this type of proof is called *structural induction*. In particular, notice the identical terminology.

structural induction

Example. Prove that every non-empty binary tree has one more node than edge.

Proof. As before, we need to define a predicate to nail down exactly what it is we'd like to prove. However, unlike all of the previous predicates we've seen, which have been boolean functions on *natural numbers*, now the domain of the predicate is the set of all non-empty binary trees.

Predicate: $P(T)$: T has one more node than edge.

Note that here the *domain* of the predicate is NOT $\mathbb{N}$, but instead the set of non-empty binary trees.

We will use structural induction to prove that for *every* non-empty binary tree T, $P(T)$ holds.

Base Case: Our base case is determined by the first rule. Suppose T is a single node. Then it has one node and no edges, so $P(T)$ holds.

Induction Step: We'll divide our proof into two parts, one for each recursive rule.

- Let T_1 and T_2 be two non-empty binary trees, and assume $P(T_1)$ and $P(T_2)$ hold. (This is the *induction hypothesis*.) Let T be the tree constructed by attaching a node r to the roots of T_1 and T_2. Let $V(G)$ and $E(G)$ denote the number of nodes and edges in a tree G, respectively. Then we have the equations

$$V(T) = V(T_1) + V(T_2) + 1$$
$$E(T) = E(T_1) + E(T_2) + 2$$

since one extra node (new root r) and two extra edges (from r to the roots of T_1 and T_2) were added to form T. **By the induction hypothesis**, $V(T_1) = E(T_1) + 1$ and $V(T_2) = E(T_2) + 1$, and so

$$\begin{aligned} V(T) &= E(T_1) + 1 + E(T_2) + 1 + 1 \\ &= E(T_1) + E(T_2) + 2 + 1 \\ &= E(T) + 1 \end{aligned}$$

Therefore $P(T)$ holds.

- Let T_1 be a non-empty binary tree, and suppose $P(T_1)$ holds. Let T be the tree formed by taking a new node r and adding an edge to the root of T_1. Then $V(T) = V(T_1) + 1$ and $E(T) = E(T_1) + 1$, and since $V(T_1) = E(T_1) + 1$ (**by the induction hypothesis**), we have

$$V(T) = E(T_1) + 2 = E(T) + 1. \quad \blacksquare$$

In structural induction, we identify some property that is satisfied by the simplest (base) elements of the set, and then show that the property is *preserved* under each of the recursive construction rules.

We say that such a property is *invariant* under the recursive rules, meaning it isn't affected when the rules are applied. The term "invariant" will reappear throughout this course in different contexts.

Here is some intuition: imagine you have a set of Lego blocks. Starting with individual Lego pieces, there are certain "rules" that you can use to combine Lego objects to build larger and larger structures, corresponding to (say) different ways of attaching Lego pieces together. This is a recursive way of describing the (infinite!) set of all possible Lego creations.

Now suppose you'd like to make a perfectly spherical object, like a soccer ball or the Death Star. Unfortunately, you look in your Lego kit and all you see are rectangular pieces! Naturally, you complain to your mother (who bought the kit for you) that you'll be unable to make a perfect sphere using the kit. But she remains unconvinced: maybe you should try doing it, she suggests, and if you're lucky you'll come up with a clever way of arranging the pieces to make a sphere. Aha! This is *impossible*, since you're starting with non-spherical pieces, and you (being a Lego expert) know that no matter which way you combine Lego objects together, starting with rectangular objects yields only other rectangular objects as results. So even though there are many, many different rectangular structures you could build, none of them could ever be perfect spheres.

A Larger Example

Let us turn our attention to another useful example of induction: proving the equivalence of recursive and non-recursive definitions. We know from our study of Python that often problems can be solved using either recursive or iterative programs, but we've taken it for granted that these programs really can accomplish the same task. We'll look later in this course at *proving* things about what programs do, but for a warm-up in this section, we'll step back from programs and prove a similar type of mathematical result.

Although often a particular problem lends itself more to one technique than the other.

Example. Consider the following recursively defined set $S \subseteq \mathbb{N} * \mathbb{N}$:

- $(0,0) \in S$
- If $(a,b) \in S$, then both $(a+1, b+1) \in S$ and $(a+3, b) \in S$

Again, there are *two* recursive rules here.

Also, define the set $S' = \{(x,y) \in \mathbb{N} * \mathbb{N} \mid x \geq y \wedge 3 \mid x - y\}$. Prove that these two definitions are equivalent, i.e., $S = S'$.

Here, $3 \mid x - y$ means that $x - y$ is divisible by 3.

Proof. We divide our solution into two parts. First, we show using structural induction that $S \subseteq S'$; that is, every element of S satisfies the property of S'. Then, we prove using complete induction that $S' \subseteq S$; that is, every element of S' can be constructed from the base and recursive rules of S.

Part 1: $S \subseteq S'$. In this part, we show that the base case of S is in S', and that all elements generated using the recursive rules of S are also in S'. For clarity, we define the predicate

$$P(x,y) : x \geq y \wedge 3 \mid x - y$$

The only base element of S is $(0,0)$. Clearly, $P(0,0)$ is true, as $0 \geq 0$ and $3 \mid 0$.

Now for the induction step. There are two recursive rules for S. Let $(a,b) \in S$, and suppose $P(a,b)$ holds. Consider $(a+1, b+1)$. By the induction hypothesis, $a \geq b$, and so $a + 1 \geq b + 1$. Also, $(a+1) - (b+1) = a - b$, which is divisible by 3 (again by the I.H.). So $P(a+1, b+1)$ also holds.

Finally, consider $(a+3, b)$. Since $a \geq b$ (by the I.H.), $a + 3 \geq b$. Also, since $3 \mid a - b$ (again by the I.H.), we can write $a - b = 3k$. Then $(a+3) - b = 3(k+1)$, so $3 \mid (a+3) - b$. Therefore, $P(a+3, b)$ holds.

Part 2: $S' \subseteq S$. We would like to use complete induction, but we can only apply that technique to natural numbers, and not pairs of natural

numbers. So we need to associate each pair (a, b) with a single natural number. We can do this by considering the *sum* of the pair. We define the following predicate:

$$P(n): \text{ for every } (x,y) \in S' \text{ such that } x+y=n,\ (x,y) \in S.$$

It should be clear that proving $\forall n \in \mathbb{N}, P(n)$ is equivalent to proving that $S' \subseteq S$. We will prove the former using complete induction.

The base case is $n = 0$. the only element of S' whose (x, y) sums to o is $(0, 0)$, which is certainly in S by the base rule of the recursive definition.

Now let $k \in \mathbb{N}$, and suppose $P(0), P(1), \ldots, P(k)$ all hold. Let $(x, y) \in S'$ such that $x + y = k + 1$. We will prove that $(x, y) \in S$. There are two cases to consider:

- $y > 0$. Then since $x \geq y$, $x > 0$. Then $(x-1, y-1) \in S'$, and $(x-1)+(y-1) = k-1$. By the **Induction Hypothesis** (in particular, $P(k-1)$), $(x-1, y-1) \in S$. Then $(x, y) \in S$ by applying the first recursive rule in the definition of S.
- $y = 0$. Since $k + 1 > 0$, it must be the case that $x > 0$. Then since $x - y = x$, x must be divisible by 3, and so $x \geq 3$. Then $(x-3, y) \in S'$ and $(x-3)+y = k-2$, so by the **Induction Hypothesis** (in particular, $P(k-2)$), $(x-3, y) \in S$. Applying the second recursive rule in the definition of S shows that $(x, y) \in S$. ■

The > 0 checks ensure that $x - 1, y - 1 \in \mathbb{N}$.

Exercises

1. Prove that for all $n \in \mathbb{N}$,
$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$
2. Let $a \in \mathbb{R}$, $a \neq 1$. Prove that for all $n \in \mathbb{N}$,
$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}.$$
3. Prove that for all $n \geq 1$,
$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1}.$$
4. Prove that $\forall n \in \mathbb{N}$, the units digit of 4^n is either 1, 4, or 6.

5. Prove that $\forall n \in \mathbb{N}, 3 \mid 4^n - 1$, where "$m \mid n$" means that m divides n, or equivalently that n is a multiple of m. This can be expressed algebraically as $\exists k \in \mathbb{N}, n = mk$.
6. Prove that for all $n \geq 2$, $2^n + 3^n < 4^n$.
7. Let $m \in \mathbb{N}$. Prove that for all $n \in \mathbb{N}$, $m \mid (m+1)^n - 1$.
8. Prove that $\forall n \in \mathbb{N}, n^2 \leq 2^n + 1$. **Hint:** first prove, without using induction, that $2n + 1 \leq n^2 - 1$ for $n \geq 3$.
9. Find a natural number $k \in \mathbb{N}$ such that for all $n \geq k$, $n^3 + n < 3^n$. Then, prove this result using simple induction.
10. Prove that $3^n < n!$ for all $n > 6$.
11. Prove that for every $n \in \mathbb{N}$, every set of size n has exactly 2^n subsets.
12. Find formulas for the number of *even-sized* subsets and *odd-sized* subsets of a set of size n. Prove that your formulas are correct in a single induction argument.

    So your predicate should be something like "every set of size n has ... even-sized subsets and ... odd-sized subsets."
13. Prove, using either simple or complete induction, that any binary string begins and ends with the same character *if and only if* it contains an even number of occurrences of substrings from $\{01, 10\}$.

    A binary string is a string containing only 0's and 1's.
14. A *ternary tree* is a tree where each node has at most 3 children. Prove that for every $n \geq 1$, every non-empty ternary tree of height n has at most $(3^n - 1)/2$ nodes.
15. Let $a \in \mathbb{R}$, $a \neq 1$. Prove that for all $n \in \mathbb{N}$,

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

    Challenge: can you mathematically derive this formula by starting from the standard geometric identity?
16. Recall two standard trigonometric identities:

$$\cos(x+y) = \cos(x)\cos(y) - \sin(x)\sin(y)$$
$$\sin(x+y) = \sin(x)\cos(y) + \cos(x)\sin(y)$$

    Also recall the definition of the imaginary number $i = \sqrt{-1}$. Prove, using induction, that

$$(\cos(x) + i\sin(x))^n = \cos(nx) + i\sin(nx).$$

17. The *Fibonacci* sequence is an infinite sequence of natural numbers $f_1, f_2, \ldots$ with the following recursive definition:

$$f_i = \begin{cases} 1, & \text{if } i = 1, 2 \\ f_{i-1} + f_{i-2}, & \text{if } i > 2 \end{cases}$$

(a) Prove that for all $n \geq 1$, $\sum_{i=1}^{n} f_i = f_{n+2} - 1$.

(b) Prove that for all $n \geq 1$, $\sum_{i=1}^{n} f_{2i-1} = f_{2n}$.

(c) Prove that for all $n \geq 2$, $f_n^2 - f_{n+1}f_{n-1} = (-1)^{n-1}$.

(d) Prove that for all $n \geq 1$, $\gcd(f_n, f_{n+1}) = 1$.

You may use the fact that for all $a < b$, $\gcd(a, b) = \gcd(a, b - a)$.

(e) Prove that for all $n \geq 1$, $\sum_{i=1}^{n} f_i^2 = f_n f_{n+1}$.

18. A *full binary tree* is a non-empty binary tree where every node has exactly 0 or 2 children. Equivalently, every internal node (non-leaf) has exactly two children.

(a) Prove using complete induction that every full binary tree has an odd number of nodes.

(b) Prove using complete induction that every full binary tree has exactly one more leaf than internal nodes.

You can choose to do induction on either the height or number of nodes in the tree. A solution with simple induction is also possible, but less generalizable.

(c) Give a recursive definition for the set of all full binary trees.

(d) Reprove parts (a) & (b) using structural induction instead of complete induction.

19. Consider the sets of binary trees with the following property: for each node, the heights of its left and right children differ by at most 1. Prove that every binary tree *with this property* of height n has at least $(1.5)^n - 1$ nodes.

20. Let $k > 1$. Prove that for all $n \in \mathbb{N}$, $\left(1 - \frac{1}{k}\right)^n \geq 1 - \frac{n}{k}$.

21. Consider the following recursively defined function $f : \mathbb{N} \to \mathbb{N}$.

$$f(n) = \begin{cases} 2, & \text{if } n = 0 \\ 7, & \text{if } n = 1 \\ (f(n-1))^2 - f(n-2), & \text{if } n \geq 2 \end{cases}$$

Prove that for all $n \in \mathbb{N}$, $3 \mid f(n) - 2$. It will be helpful to phrase your predicate here as $\exists k \in \mathbb{N}, f(n) = 3k + 2$.

22. Prove that every natural number greater than 1 can be written as the sum of prime numbers.

23. We define the set S of strings over the alphabet $\{[,]\}$ recursively by

- ϵ, the empty string, is in S
- If $w \in S$, then so is $[w]$
- If $x, y \in S$, then so is xy

Prove that every string in S is *balanced*, i.e., the number of left brackets equals the number of right brackets.

24. The *Fibonacci* trees T_n are a special set of binary trees defined recursively as follows.

    - T_1 and T_2 are binary trees with only a single node.
    - For $n > 2$, T_n consists of a root node whose left subtree is T_{n-1}, and whose right subtree is T_{n-2}.

    (a) Prove that for all $n \geq 2$, the height of T_n is $n - 2$.
    (b) Prove that for all $n \geq 1$, T_n has f_n leaves, where f_n is the n-th Fibonacci number.

25. Consider the following recursively defined set $S \subset \mathbb{N}^2$.

    - $2 \in S$
    - If $k \in S$, then $k^2 \in S$
    - If $k \in S$, and $k \geq 2$, then $\frac{k}{2} \in S$

    (a) Prove that every element of S is a power of 2, i.e., can be written in the form 2^m for some $m \in \mathbb{N}$.
    (b) Prove that every power of 2 (including 2^0) is in S.

26. Consider the set $S \subset \mathbb{N}^2$ of ordered pairs of integers defined by the following recursive definition:

    - $(3, 2) \in S$
    - If $(x, y) \in S$, then $(3x - 2y, x) \in S$

    Also consider the set $S' \subset \mathbb{N}^2$ with the following non-recursive definition:

    $$S' = \{(2^{k+1} + 1, 2^k + 1) \mid k \in \mathbb{N}\}.$$

    Prove that $S = S'$, or in other words, that the recursive and non-recursive definitions agree.

27. We define the set of propositional formulas PF as follows:

    - Any proposition P is a propositional formula.
    - If F is a propositional formula, then so is $\neg F$.
    - If F_1 and F_2 are propositional formulas, then so are $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$, and $F_1 \Leftrightarrow F_2$.

    Prove that for all propositional formulas F, F has a logically equivalent formula G such that G only has negations applied to propositions. For example, we have the equivalence

    $$\neg(\neg(P \wedge Q) \Rightarrow R) \Longleftrightarrow (\neg P \vee \neg Q) \wedge \neg R$$

    Hint: you won't have much luck applying induction directly to the statement in the question. (Try it!) Instead, prove the stronger statement: "F and $\neg F$ have equivalent formulas that only have negations applied to propositions."

28. It is well-known that Facebook friendships are the most important relationships you will have in your lifetime. For a person x on Facebook, let f_x denote the number of friends x has. Find a relationship between the total number of Facebook friendships in the world, and the *sum* of all of the f_x's (over every person on Facebook). Prove your relationship using induction.

29. Consider the following 1-player game. We start with n pebbles in a pile, where $n \geq 1$. A *valid move* is the following: pick a pile with more than 1 pebble, and divide it into two smaller piles. When this happens, add to your score the *product* of the sizes of the two new piles. Continue making moves until no more can be made, i.e., there are n piles each containing a single pebble.

    Prove using complete induction that no matter how the player makes her moves, she will *always score* $\dfrac{n(n-1)}{2}$ points when playing this game with n pebbles.

    So this game is completely determined by the starting conditions, and not at all by the player's choices. Sounds fun.

30. A certain summer game is played with n people, each carrying one water balloon. The players walk around randomly on a field until a buzzer sounds, at which point they stop. You may assume that when the buzzer sounds, each player has a unique closest neighbour. After stopping, each player then throws his water balloon at their closest neighbour. The winners of the game are the players who are dry after the water balloons have been thrown (assume everyone has perfect aim).

    Prove that for every odd n, this game always has at least one winner.

The following problems are for the more mathematically-inclined students.

1. The *Principle of Double Induction* is as follows. Suppose that $P(x, y)$ is a predicate with domain $\mathbb{N}^2$ satisfying the following properties:

    (1) $P(0, y)$ holds for all $y \in \mathbb{N}$

    (2) For all $(x, y) \in \mathbb{N}^2$, if $P(x, y)$ holds, then so does $P(x + 1, y)$.

    Then we may conclude that for all $x, y \in \mathbb{N}, P(x, y)$.

    Prove that the Principle of Double Induction is equivalent to the Principle of Simple Induction.

2. Prove that for all $n \geq 1$, and positive real numbers $x_1, \ldots, x_n \in \mathbb{R}^+$,

$$\frac{1 - x_1}{1 + x_1} \times \frac{1 - x_2}{1 + x_2} \times \cdots \times \frac{1 - x_n}{1 + x_n} \geq \frac{1 - S}{1 + S},$$

    where $S = \sum_{i=1}^{n} x_i$.

3. A *unit fraction* is a fraction of the form $\frac{1}{n}$, $n \in \mathbb{Z}^+$. Prove that every rational number $0 < \frac{p}{q} < 1$ can be written as the sum of *distinct* unit fractions.

Recursion

Now, programming! In this chapter, we will apply what we've learned about induction to study recursive algorithms. In particular, we will learn how to analyse the time complexity of recursive programs, for which the runtime on an input of size n depends on the runtime on smaller inputs. Unsurprisingly, this is tightly connected to the study of recursively defined (mathematical) functions; we will discuss how to go from a recurrence relation like $f(n+1) = f(n) + f(n-1)$ to a *closed form* expression like $f(n) = 2^n + n^2$. For recurrences of a special form, we will see how the *Master Theorem* gives us immediate, tight asymptotic bounds. These recurrences will be used for *divide-and-conquer* algorithms; you will gain experience with this common algorithmic paradigm and even design algorithms of your own.

Recall that asymptotic bounds involve Big-O, and are less precise than exact expressions.

Measuring Runtime

Recall that one of the most important properties of an algorithm is how long it takes to run. We can use the *number of steps* as a measurement of running time; but reporting an absolute number like "10 steps" or "1 000 000 steps" an algorithm takes is pretty meaningless unless we know how "big" the input was, since of course we'd expect algorithms to take longer on larger inputs. So a more meaningful measure of runtime is "10 steps when the input has size 2" or "1 000 000 steps when the input has size 300" or even better, "$n^2 + 2$ steps when the input has size n." But as you probably remember from CSC148, counting an exact number of steps is often tedious and arbitrary, so we care more about the *Big-O* (asymptotic) analysis of an algorithm.

In this course, we will mainly care about the upper bound on the *worst-case* runtime of algorithms; that is, the absolute longest an algorithm could run on a given input size n.

In CSC148 and earlier in this course, you analysed the runtime of iterative algorithms. As we've mentioned several times by now, induction is very similar to recursion; since induction has been the key idea of the course so far, it should come as no surprise that we'll turn our

attention to *recursive* algorithms now!

A Simple Recursive Function

Consider the following simple recursive function, which you probably saw in CSC148:

All code in this course will be in Python-like pseudocode. The syntax and methods will be mostly Python, with some English making the code more readable and/or intuitive. We'll expect you to follow a similar style.

```
1 def fact(n):
2   if n == 1:
3     return 1
4   else:
5     return n * fact(n-1)
```

In CSC148, it was surely claimed that this function computes $n!$ and we'll see later in this course how to formally *prove* that this is what the function does.

How would you informally analyse the runtime of this algorithm? One might say that the recursion depth is n, and at each call there is just one step other than the recursive call, so the runtime is $\mathcal{O}(n)$, i.e., linear time. But in performing a more thorough step-by-step analysis, we reach a stumbling block with the recursive call `fact(n-1)`: the runtime of `fact` on input n depends on its runtime on input $n-1$! Let's see how to deal with this relationship, using mathematics and induction.

Recursively Defined Functions

You should all be familiar with standard function notation: $f(n) = n^2$, $f(n) = n \log n$, or the slightly more unusual (but no less meaningful) $f(n) =$ "the number of distinct prime factors of n." There is a second way of defining functions using recursion, e.g.,

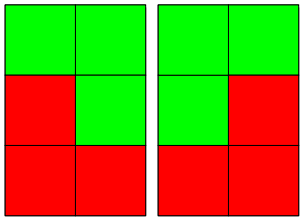
$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ f(n-1) + 2n - 1, & \text{if } n \geq 1 \end{cases}$$

recursively defined function

Recursive definitions allow us to capture marginal or relative difference between function values, even when we don't know their exact values. But recursive definitions have a significant downside: we can only calculate large values of f by computing smaller values first. For calculating extremely large, or *symbolic*, values of f (like $f(n^2 + 3n)$), a recursive definition is inadequate; what we would really like is a *closed form* expression for f, one that doesn't depend on other values of f. In this case, the closed form expression for f is $f(n) = n^2$.

You will prove this in the Exercises.

Before returning to our earlier factorial example, let us see how to apply this to a more concrete example.



Example. There are exactly two ways of tiling a 3-by-2 grid using triominoes, shown to the right:

Develop a recursive definition for $f(n)$, the number of ways of tiling a 3-by-n grid using triominoes for $n \geq 1$. Then, find a closed form expression for f.

Solution:
Note that if $n = 1$, there are *no* possible tilings, since no triomino will fit in a 3-by-1 board. We have already observed that there are 2 tilings for $n = 2$. Suppose $n > 2$. The key idea to get a recurrence is that for a 3-by-n block, first consider the upper-left square. In any tiling, there are only two possible triomino placements that can cover it (these orientations are shown in the diagram above). Once we have fixed one of these orientations, there is only one possible triomino orientation that can cover the bottom-left square (again, these are the two orientations shown in the figure).

So there are exactly two possibilities for covering both the bottom-left and top-left squares. But once we've put these two down, we've tiled the leftmost 3-by-2 part of the grid, and the remainder of the tiling really just tiles the remaining 3-by-$(n-2)$ part of the grid; there are $f(n-2)$ such tilings. Since these two parts are independent of each other, we get the total number of tilings by *multiplying* the number of possibilities for each. Therefore the recurrence relation is:

Because we've expressed $f(n)$ in terms of $f(n-2)$, we need *two* base cases – otherwise, at $n = 2$ we would be stuck, as $f(0)$ is undefined.

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2f(n-2), & \text{if } n > 2 \end{cases}$$

Now that we have the recursive definition of f, we would like to find its closed form expression. The first step is to *guess* the closed form expression, by a "brute force" approach known as *repeated substitution*. Intuitively, we'll expand out the recursive definition until we find a pattern.

So much of mathematics is finding patterns.

$$\begin{aligned} f(n) &= 2f(n-2) \\ &= 4f(n-4) \\ &= 8f(n-6) \\ &\vdots \\ &= 2^k f(n-2k) \end{aligned}$$

There are two possibilities. If n is odd, say $n = 2m + 1$, then we have $f(n) = 2^m f(1) = 0$, since $f(1) = 0$. If n is even, say $n = 2m$, then $f(n) = 2^{m-1} f(2) = 2^{m-1} \cdot 2 = 2^m$. Writing our final answer in terms of n only:

$$f(n) = \begin{cases} 0, & \text{if } n \text{ is odd} \\ 2^{\frac{n}{2}}, & \text{if } n \text{ is even} \end{cases}$$

Thus we've obtained a closed form formula $f(n)$ – except the $\vdots$ in our repeated substitution does not constitute a formal proof! When you saw the $\vdots$, you probably interpreted it as "repeat over and over again until..." and we already know how to make this thinking formal: induction! That is, given the *recursive definition* of f, we can prove using complete induction that $f(n)$ has the closed form given above. This is a rather straightforward argument, and we leave it for the Exercises.

Why complete and not simple induction? We need the induction hypothesis to work for $n - 2$, and not just $n - 1$.

We will now apply this technique to our earlier example.

Example. Analyse the asymptotic worst-case running time of `fact(n)`, in terms of n.

Solution:
Let $T(n)$ denote the worst-case running time of `fact` on input n. In this course, we will ignore exact step counts entirely, replacing these counts with constants.

The *base case* of this method is when $n = 1$; in this case, the `if` block executes and the method returns `1`. This is done in constant time, and so we can say that $T(1) = c$ for some constant c.

Constant always means "independent of input size."

What if $n > 1$? Then `fact` makes a recursive call, and to analyse the runtime we consider the recursive and non-recursive parts separately. The non-recursive part is simple: only a constant number of steps occur (the `if` check, multiplication by n, and the `return`), so let's say the non-recursive part takes d steps. What about the recursive part? The recursive call is `fact(n-1)`, which has worst-case runtime $T(n - 1)$, by definition! Therefore when $n > 1$ we get the recurrence relation $T(n) = T(n - 1) + d$. Putting this together with the base case, we get the full recursive definition of T:

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T(n-1) + d, & \text{if } n > 1 \end{cases}$$

Now, we would like to say that $T(n) = \mathcal{O}(??)$, but to do so, we really need a closed form definition of T. Once again, we use repeated

substitution.

$$\begin{aligned} T(n) &= T(n-1) + d \\ &= \big(T(n-2) + d\big) + d = T(n-2) + 2d \\ &= T(n-3) + 3d \\ &\vdots \\ &= T(1) + (n-1)d \\ &= c + (n-1)d \qquad \text{(Since } T(1) = c\text{)} \end{aligned}$$

Thus we've obtained the closed form formula $T(n) = c + (n-1)d$, modulo the $\vdots$. As in the previous example, we leave proving this closed form as an exercise.

After proving this closed form, the final step is simply to convert this closed form into an asymptotic bound on T. Since c and d are constants with respect to n, we have that $T(n) = \mathcal{O}(n)$.

Now let's see a more complicated recursive function.

Example. Consider the following code for binary search.

```
1  def bin_search(A, x):
2    '''
3    Pre: A is a sorted list (non-decreasing order).
4    Post: Returns True if and only if x is in A.
5    '''
6    if len(A) == 0:
7      return False
8    else if len(A) == 1:
9      return A[0] == x
10   else:
11     m = len(A) // 2  # Rounds down, like floor
12     if x <= A[m-1]:
13       return bin_search(A[0..m-1], x)
14     else:
15       return bin_search(A[m..len(A)-1], x)
```

One notable difference from Python is how we'll denote sublists. Here, we use the notation `A[i..j]` to mean the slice of the list `A` from index `i` to index `j`, *including* `A[i]` and `A[j]`.

We analyse the runtime of `bin_search` in terms of n, the length of the input list A. If $n = 0$ or $n = 1$, `bin_search(A,x)` takes constant time (note that it doesn't matter whether the constant is the same or different for 0 and 1).

What about when $n > 1$? Then some recursive calls are made, and we again look at the recursive and non-recursive steps separately. We

include the computation of `A[0..m-1]` and `A[m..len(A)-1]` in the non-recursive part, since argument evaluation happens before the recursive call begins.

Interestingly, this is not the case in some programming languages – an alternative is "lazy evaluation."

> **IMPORTANT ANNOUNCEMENT 1**: We will interpret all list slicing operations $A[i..j]$ as constant time, even when i and j depend on the length of the list. See the discussion in the following section.

Interpreting list slicing as constant time, the non-recursive cost of `bin_search` is constant time. What about the recursive calls? In all possible cases, only one recursive call occurs. What is the size of the list of the recursive call? When either recursive call happens, $m = \left\lfloor \frac{n}{2} \right\rfloor$, meaning the recursive call either is on a list of size $\left\lfloor \frac{n}{2} \right\rfloor$ or $\left\lceil \frac{n}{2} \right\rceil$.

> **IMPORTANT ANNOUNCEMENT 2**: In this course, we won't care about floors and ceilings. We'll always assume that the input sizes are "nice" so that the recursive calls always divide the list evenly. In the case of binary search, we'll assume that n is a power of 2.

You may look in Vassos Hadzilacos' course notes for a complete handling of floors and ceilings. The algebra is a little more involved, but the bottom line is that the asymptotic analysis is unchanged.

With this in mind, we conclude that the recurrence relation for $T(n)$ is $T(n) = T\left(\frac{n}{2}\right) + d$. Therefore the full recursive definition of T is

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ T\left(\frac{n}{2}\right) + d, & \text{if } n > 1 \end{cases}$$

Again, we omit floors and ceilings.

Let us use repeated substitution to guess a closed form. Assume that $n = 2^k$ for some natural number k.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + d \\ &= \left(T\left(\frac{n}{4}\right) + d\right) + d = T\left(\frac{n}{4}\right) + 2d \\ &= T\left(\frac{n}{8}\right) + 3d \\ &\vdots \\ &= T\left(\frac{n}{2^k}\right) + kd \\ &= T(1) + kd && \text{(Since } n = 2^k\text{)} \\ &= c + kd && \text{(Since } T(1) = c\text{)} \end{aligned}$$

Once again, we'll leave proving this closed form to the Exercises. So $T(n) = c + kd$. This expression is quite misleading, because it seems to not involve an n, and hence be constant time – which we know is not the case for binary search! The key is to remember that $n = 2^k$, so $k = \log_2 n$. Therefore we have $T(n) = c + d \log_2 n$, and so $T(n) = \mathcal{O}(\log n)$.

Aside: List Slicing vs. Indexing

In our analysis of binary search, we assumed that the list slicing operation `A[0..m-1]` took constant time. However, this is not the case in Python and many other programming languages, which implement this operation by copying the sliced elements into a new list. Depending on the scale of your application, this can be undesirable for two reasons: this copying takes time *linear* in the size of the slice, and uses *linear* additional memory.

While we are not so concerned in this course about the second issue, the first can drastically change our runtime analysis (e.g., in our analysis of binary search). However, there is always another way to implement these algorithms without this sort of slicing that can be done in constant time, and without creating new lists. The key idea is to use variables to keep track of the start and end points of the section of the list we are interested in, but keep the whole list all the way through the computation. We illustrate this technique in our modified binary search:

```
def indexed_bin_search(A, x, first, last):
  if first > last:
    return False
  else if first == last:
    return A[first] == x
  else:
    m = (first + last + 1) // 2
    if x <= A[m-1]:
      return indexed_bin_search(A, x, first, m - 1)
    else:
      return indexed_bin_search(A, x, m, last)
```

In this code, the same list `A` is passed to each recursive call; the range of searched values, on the other hand, indeed gets smaller, as

the first and last parameters change. More technically, the size of the range, `last - first + 1`, decreases by a (multiplicative) factor of two at each recursive call.

Passing indices as arguments works well for recursive functions that work on smaller and smaller segments of a list. We've just introduced the most basic version of this technique. However, many other algorithms involve making new lists in more complex ways, and it is usually possible to make these algorithms *in-place*, i.e., to use a constant amount of extra memory, and do operations by changing the elements of the original list.

Because we aren't very concerned with this level of implementation detail in this course, we'll use the shortcut of interpreting list slicing as taking constant time, *keeping in mind actual naïve implementations take linear time*. This will allow us to perform our runtime analyses without getting bogged down in clever implementation tricks.

A Special Recurrence Form

In general, finding exact closed forms for recurrences can be tedious or even impossible. Luckily, we are often not really looking for a closed form solution to a recurrence, but an *asymptotic* bound. But even for this relaxed goal, the only method we have so far is to find a closed form and turn it into an asymptotic bound. In this section, we'll look at a powerful technique with the caveat that it works only for a special recurrence form.

We can motivate this recurrence form by considering a style of recursive algorithm called *divide-and-conquer*. We'll discuss this in detail in the next section, but for now consider the `mergesort` algorithm, which can roughly be outlined in three steps:

mergesort

1. Divide the list into two equal halves.
2. Sort each half separately, using recursive calls to mergesort.
3. Merge each of the sorted halves.

```
1 def mergesort(A):
2   if len(A) == 1:
3     return A
4   else:
5     m = len(A) // 2
6     L1 = mergesort(A[0..m-1])
```

```
    L2 = mergesort(A[m..len(A)-1])
    return merge(L1, L2)

def merge(A, B):
  i = 0
  j = 0
  C = []
  while i < len(A) and j < len(B):
    if A[i] <= B[j]:
      C.append(A[i])
      i += 1
    else:
      C.append(B[j])
      j += 1
  return C + A[i..len(A)-1] + B[j..len(B)-1] # List concatenation
```

Consider the analysis of $T(n)$, the runtime of `mergesort(A)` where `len(A) = n`. Steps 1 and 3 together take linear time. What about Step 2? Since this is the recursive step, we'd expect T to appear here. There are two fundamental questions:

Careful implementations of mergesort can do step 1 in constant time, but merging always takes linear time.

- What is the number of recursive calls?
- What is the size of the lists passed to the recursive calls?

From the written description, you should be able to intuit that there are *two* recursive calls, each on a list of size $\frac{n}{2}$. So the cost of Step 2 is $2T\left(\frac{n}{2}\right)$. Putting all three steps together, we get a recurrence relation

For the last time, we'll point out that we ignore floors and ceilings.

$$T(n) = 2T\left(\frac{n}{2}\right) + cn.$$

This is an example of our *special recurrence form*:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where $a, b \in \mathbb{Z}^+$ are constants and $f : \mathbb{N} \to \mathbb{R}^+$ is some arbitrary function.

Though we'll soon restrict f to ease the analysis of this recursive form.

Before we get to the Master Theorem, which gives us an immediate asymptotic bound for recurrences of this form, let's discuss some intuition. The special recurrence form has three parameters: a, b, and f. Changing how big they are affects the overall runtime:

- a is the "number of recursive calls"; the bigger a is, the more recursive calls, and the bigger we expect $T(n)$ to be.

- b determines the rate of decrease of the problem size; the larger b is, the faster the problem size goes down to 1, and the smaller $T(n)$ is.
- $f(n)$ is the cost of the non-recursive part; the bigger $f(n)$ is, the bigger $T(n)$ is.

We can further quantify this relationship by considering the following even more specific form:

$$f(n) = \begin{cases} c, & \text{if } n = 1 \\ af\left(\frac{n}{b}\right) + n^k, & \text{if } n > 1 \end{cases}$$

Suppose n is a power of b, say $n = b^r$. Using repeated substitution,

$$\begin{aligned} f(n) &= af\left(\frac{n}{b}\right) + n^k \\ &= a\left(af\left(\frac{n}{b^2}\right) + \frac{n^k}{b^k}\right) + n^k = a^2 f\left(\frac{n}{b^2}\right) + n^k\left(1 + \frac{a}{b^k}\right) \\ &= a^3 f\left(\frac{n}{b^3}\right) + n^k\left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2\right) \\ &\vdots \\ &= a^r f\left(\frac{n}{b^r}\right) + n^k \sum_{i=0}^{r-1} \left(\frac{a}{b^k}\right)^i \\ &= a^r f(1) + n^k \sum_{i=0}^{r-1} \left(\frac{a}{b^k}\right)^i && \text{(Since } n = b^r\text{)} \\ &= ca^r + n^k \sum_{i=0}^{r-1} \left(\frac{a}{b^k}\right)^i \\ &= cn^{\log_b a} + n^k \sum_{i=0}^{r-1} \left(\frac{a}{b^k}\right)^i \end{aligned}$$

Note that $r = \log_b n$, and so $a^r = a^{\log_b n} = b^{\log_b a \cdot \log_b n} = n^{\log_b a}$.

The latter term looks like a geometric series, for which we may use our geometric series formula. However, this only applies when the common ratio $\frac{a}{b^k}$ is not equal to 1. Therefore, there are two cases.

- Case 1: $\frac{a}{b^k} = 1$, so $a = b^k$. Taking logs, we have $\log_b a = k$. In this case, the expression becomes

$$\begin{aligned} f(n) &= cn^k + n^k \sum_{i=0}^{r-1} 1^i \\ &= cn^k + n^k r \\ &= cn^k + n^k \log_b n \\ &= \mathcal{O}(n^k \log n) \end{aligned}$$

- Case 2: $a \neq b^k$. Then by the geometric series formula,

$$\begin{aligned} f(n) &= cn^{\log_b a} + n^k \left(\frac{1 - \frac{a^r}{b^{kr}}}{1 - \frac{a}{b^k}} \right) \\ &= cn^{\log_b a} + n^k \left(\frac{1 - \frac{n^{\log_b a}}{n^k}}{1 - \frac{a}{b^k}} \right) \\ &= \left(c - \frac{1}{1 - \frac{a}{b^k}} \right) n^{\log_b a} + \left(\frac{1}{1 - \frac{a}{b^k}} \right) n^k \end{aligned}$$

  There are two occurrences of n in this expression: $n^{\log_b a}$ and n^k. Asymptotically, the higher exponent dominates; so if $\log_b a > k$, then $f(n) = \mathcal{O}(n^{\log_b a})$, and if $\log_b a < k$, then $f(n) = \mathcal{O}(n^k)$.

With this intuition in mind, let us now state the Master Theorem.

Master Theorem

Theorem (Master Theorem)**.** *Let $T : \mathbb{N} \to \mathbb{R}^+$ be a recursively defined function with recurrence relation $T(n) = aT(n/b) + f(n)$, for some constants $a, b \in \mathbb{Z}^+$, $b > 1$, and $f : \mathbb{N} \to \mathbb{R}^+$. Furthermore, suppose $f(n) = \Theta(n^k)$ for some $k \geq 0$. Then we can conclude the following about the asymptotic complexity of T:*

(1) If $k = \log_b a$, then $T(n) = \mathcal{O}(n^k \log n)$.
(2) If $k < \log_b a$, then $T(n) = \mathcal{O}(n^{\log_b a})$.
(3) If $k > \log_b a$, then $T(n) = \mathcal{O}(n^k)$.

Let's see some examples of the Master Theorem in action.

Example. Consider the recurrence for mergesort: $T(n) = 2T(n/2) + dn$. Here $a = b = 2$, so $\log_2 2 = 1$, while $dn = \Theta(n^1)$. Therefore Case 1 of the Master Theorem applies, and $T(n) = \mathcal{O}(n \log n)$, as we expected.

Example. Consider the recurrence $T(n) = 49T(n/7) + 50n^\pi$. Here, $\log_7 49 = 2$ and $2 < \pi$, so by Case 3 of the Master Theorem, $T(n) = \mathcal{O}(n^\pi)$.

Even though the Master Theorem is useful in a lot of situations, be sure you understand the statement of the theorem to see exactly when it applies (see Exercises for some questions investigating this).

Divide-and-Conquer Algorithms

Now that we have seen the Master Theorem, let's discuss some algorithms for which it can help us analyse the runtime! A key fea-

ture of the recurrence form $aT(n/b) + f(n)$ is that *each of the recursive calls has the same size*. This naturally leads to the **divide-and-conquer** paradigm, which can be summarized as follows:

divide-and-conquer
An algorithmic *paradigm* is a general strategy for designing algorithms to solve problems. You will see many more such strategies in CSC373.

```
def divide-and-conquer(P):
  if P has "small enough" size:
    return solve_directly(P)
  else:
    divide P into smaller problems P_1, ..., P_k (same size)
    for i from 1 to k:
      # Solve each subproblem recursively
      s_i = divide_and_conquer(P_i)
    # combine the s_1 ... s_k to solve P
    return combine(s_1 ... s_k)
```

This is a very general template — in fact, it may seem exactly like your mental model of recursion so far, and certainly it is a recursive strategy. What distinguishes divide-and-conquer algorithms from a lot of other recursive procedures is that we divide the problem into *two or more* parts and solve the subproblems for each part, whereas recursive functions in general may make only a single recursive call, like in `fact` or `bin_search`.

Another common non-divide-and-conquer recursive design pattern is taking a list, processing the first element, then recursively processing the rest of the list (and combining the results).

This introduction to the divide-and-conquer paradigm was deliberately abstract. However, we have already discussed one divide-and-conquer algorithm: mergesort! Let us now see two more examples of divide-and-conquer algorithms: fast multiplication and quicksort.

Fast Multiplication

Consider an algorithm for multiplying two numbers: 1234×5678. We might start by writing this as

$$(1 * 1000 + 2 * 100 + 3 * 10 + 4) * (5 * 1000 + 6 * 100 + 7 * 10 + 8)$$

Expanding this product requires 16 one-digit multiplications ($1 * 5, 2 * 5, 3 * 5, 4 * 5, 1 * 6, \ldots, 4 * 8$), and then some one-digit additions to add everything up. In general, multiplying two n-digit numbers this way requires $\mathcal{O}(n^2)$ one-digit operations.

Now, let's see a different way of making this faster. Using a *divide-and-conquer* approach, we want to split 1234 and 5678 into smaller

numbers:

$$1234 = 12 \cdot 100 + 34, \quad 5678 = 56 \cdot 100 + 78.$$

Now we use some algebra to write the product $1234 \cdot 5678$ as the *combination* of some *smaller* products:

$$\begin{aligned} 1234 \cdot 5678 &= (12 \cdot 100 + 34)(56 \cdot 100 + 78) \\ &= (12 \cdot 56) \cdot 10000 + (12 \cdot 78 + 34 \cdot 56) \cdot 100 + 34 \cdot 78 \end{aligned}$$

So now instead of multiplying 4-digit numbers, we have shown how to find the solution by multiplying some 2-digit numbers, a much easier problem! Note that we aren't counting multiplication by powers of 10, since that amounts to just adding some zeroes to the end of the numbers.

On a computer, we would use base-2 instead of base-10 to take advantage of the "adding zeros," which corresponds to (very fast) bit-shift machine operations.

Reducing 4-digit multiplication to 2-digit multiplication may not seem that impressive; but now, we'll generalize this to arbitrary n-digit numbers (the difference in multiplying 100-digit vs. 50-digit numbers may be more impressive).

Let x and y be n-digit numbers. For simplicity, assume n is a power of 2. Then we can divide x and y each into two halves:

$$x = 10^{\frac{n}{2}}a + b$$
$$y = 10^{\frac{n}{2}}c + d$$

Where a, b, c, d are $\frac{n}{2}$-digit numbers. Then

$$x \cdot y = (ac)10^n + (ad + bc)10^{\frac{n}{2}} + bd.$$

We have found a mathematical identity that seems useful, and we can use this to develop a multiplication algorithm. Let's see some pseudocode:

The *length* of a number here refers to the number of digits in its decimal representation.

```
def rec_mult(x,y):
  n = length of x # Assume x and y have the same length
  if n == 1:
    return x * y
  else:
    a = x // 10^(n//2)
    b = x % 10^(n//2)
    c = y // 10^(n//2)
    d = y % 10^(n//2)

    r = rec_mult(a, c)
    s = rec_mult(a, d)
```

```
    t = rec_mult(b, c)
    u = rec_mult(b, d)

    return r * 10^n + (s + t) * 10^(n//2) + u
```

Now, let's talk about the running time of this algorithm, in terms of the size n of the two numbers. Note that there are four recursive calls; each call multiplies two numbers of size $\frac{n}{2}$, so the cost of the recursive calls is $4T\left(\frac{n}{2}\right)$. What about the non-recursive parts? Note that the final `return` step involves addition of $2n$-digit numbers, which takes $\Theta(n)$ time. Therefore we have the recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + cn.$$

By the Master Theorem, we have $T(n) = \mathcal{O}(n^2)$.

So, this approach didn't help! We had an arguably more complicated algorithm that achieved the same asymptotic runtime as what we learned in elementary school! Moral of the story: **Divide-and-conquer, like all algorithmic paradigms, doesn't always lead to "better" solutions!**

This is a *serious* lesson. It is not the case that everything we teach you works for every situation. It is up to you to carefully put together your knowledge to figure out how to approach problems!

In the case of fast multiplication, though, we can use more math to improve the running time. Note that the "cross term" $ad + bc$ in the algorithm required two multiplications to compute naïvely; however, it is correlated with the values of ac and bd with the following straightforward identity:

$$(a+b)(c+d) = ac + (ad+bc) + bd$$
$$(a+b)(c+d) - ac - bd = ad + bc$$

So we can compute $ad + bc$ by calculating just one additional product $(a+b)(c+d)$ (together with ac and bd that we are calculating anyway). This trick underlies the multiplication algorithm of Karatsuba (1960).

```
def fast_rec_mult(x,y):
  n = length of x # Assume x and y have the same length
  if n == 1:
    return x * y
  else:
    a = x // 10^(n//2)
    b = x % 10^(n//2)
    c = y // 10^(n//2)
    d = y % 10^(n//2)
```

```
    p = fast_rec_mult(a + b, c + d)
    r = fast_rec_mult(a, c)
    u = fast_rec_mult(b, d)

    return r * 10^n + (p - r - u) * 10^(n//2) + u
```

You can study the (improved!) runtime of this algorithm in the exercises.

Quicksort

In this section, we explore the divide-and-conquer sorting algorithm known as *quicksort*, which in practice is one of the most commonly used sorting algorithms. First, we give the pseudocode for this algorithm; note that this follows a very clear divide-and-conquer pattern. Unlike `fast_rec_mult` and `mergesort`, the hard work is done in the divide (partition) step, not the combine step. The combine step for `quicksort` can be made to take a constant amount of work.

quicksort

Our naïve implementation below does list concatenation in linear time because of list slicing, but in fact a more clever implementation using indexing accomplishes this in constant time.

```
def quicksort(A):
  if len(A) <= 1:
    # do nothing (A is already sorted)
  else:
    choose some element x of A (the "pivot")
    partition (divide) the rest of the elements of A into two lists:
      - L, the elements of A <= x
      - G, the elements of A > x
    sort L and G recursively
    combine the sorted lists in the order L + [x] + G
    set A equal to the new list
```

Before moving on, an excellent exercise is to take the above pseudocode and implement quicksort yourself. As we will discuss again and again, implementing algorithms yourself is the best way to understand them. Remember that the only way to improve your coding abilities is to code lots — even something as simple and common as sorting algorithms offers great practice. See the Exercises for more examples.

```python
def quicksort(A):
  if len(A) <= 1:
    pass
  else:
    # Choose the final element as the pivot
    pivot = A[-1]

    # Partition the rest of A with respect to the pivot
    L, G = partition(A[0:-1], pivot)
    # Sort each list recursively
    quicksort(L)
    quicksort(G)

    # Combine
    sorted = L + [pivot] + G
    # Set A equal to the sorted list
    for i in range(len(A)):
      A[i] = sorted[i]

def partition(A, pivot):
  L = []
  G = []
  for x in A:
    if x <= pivot:
      L.append(x)
    else:
      G.append(x)
  return L, G
```

Let us try to analyse the running time $T(n)$ of this algorithm, where n is the length of the input list A. First, the base case $n = 1$ takes constant time. The `partition` method takes linear time, since it is called on a list of length $n - 1$ and contains a `for` loop that loops through all $n - 1$ elements. The Python list methods in the rest of the code also take linear time, though a more careful implementation could reduce this. But because partitioning the list always takes linear time, the *non*-recursive cost of `quicksort` is linear.

What about the costs of the recursive steps? There are two of them: `quicksort(L)` and `quicksort(G)`, so the recursive cost in terms of L and G is $T(|L|)$ and $T(|G|)$. Therefore a potential recurrence is:

Here $|A|$ denotes the length of the list A.

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ T(|L|) + T(|G|) + dn, & \text{if } n > 1 \end{cases}$$

What's the problem with this recurrence? It depends on what L and G

are, which in turn depends on the input array and the chosen pivot! In particular, we can't use either repeated substitution or the Master Theorem to analyse this function. In fact, the asymptotic running time of this algorithm can range from $\Theta(n \log n)$ to $\Theta(n^2)$, the latter of which is just as bad as bubblesort!

See the Exercises for details.

This begs the question: why is quicksort so used in practice? Two reasons: quicksort takes $\Theta(n \log n)$ time "on average", and careful implementations of quicksort yield better *constants* than other $\Theta(n \log n)$ algorithms like mergesort. These two facts together imply that quicksort often outperforms other sorting algorithms *in practice*!

Average-case analysis is slightly more sophisticated than what we do in this course, but you can take this to mean that most of the time, on randomly selected inputs, quicksort takes $\Theta(n \log n)$ time.

Exercises

1. Let $f : \mathbb{N} \to \mathbb{N}$ be defined as

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ f(n-1) + 2n - 1, & \text{if } n \geq 1 \end{cases}$$

   Prove using induction that the closed form for f is $f(n) = n^2$.

2. Recall the recursively defined function

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2f(n-2), & \text{if } n > 2 \end{cases}$$

   Prove that the closed form for f is

$$f(n) = \begin{cases} 0, & \text{if } n \text{ is odd} \\ 2^{\frac{n}{2}}, & \text{if } n \text{ is even} \end{cases}$$

3. Prove that the closed form expression for the runtime of `fact` is $T(n) = c + (n-1)d$.

4. Prove that the closed form expression for the runtime of `bin_search` is $T(n) = c + d \log_2 n$.

5. Let $T(n)$ be the number of binary strings of length n in which there are no consecutive 1's. So $T(0) = 1$, $T(1) = 2$, $T(2) = 3$, etc.

   (a) Develop a recurrence for $T(n)$. Hint: think about the two possible cases for the last character.

   (b) Find a closed form expression for $T(n)$.

   (c) Prove that your closed form expression is correct using induction.

6. Repeat the steps of the previous question, except with binary strings where every 1 is immediately preceded by a 0.

7. It is known that every full binary tree has an odd number of nodes. Let $T(n)$ denote the number of distinct full binary trees with n nodes. For example, $T(1) = 1$, $T(3) = 1$, and $T(7) = 5$. Give a recurrence for $T(n)$, justifying why it is correct. Then, use induction to prove that $T(n) \geq \frac{1}{n} 2^{(n-1)/2}$.

   A *full* binary tree is a binary tree where every node has either 0 or 2 children.

8. Consider the following recursively defined function

$$f(n) = \begin{cases} 3, & \text{if } n = 0 \\ 7, & \text{if } n = 1 \\ 3f(n-1) - 2f(n-2), & \text{if } n \geq 2 \end{cases}$$

   Find a closed form expression for f, and prove that it is correct using induction.

9. Consider the following recursively defined function:

$$f(n) = \begin{cases} \frac{1}{5}, & \text{if } n = 0 \\ \dfrac{1 + f(n-1)}{2}, & \text{if } n \geq 1 \end{cases}$$

   (a) Prove that for all $n \geq 1$, $f(n+1) - f(n) < f(n) - f(n-1)$.

   (b) Prove that for all $n \in \mathbb{N}$, $f(n) = 1 - \dfrac{4}{5 \cdot 2^n}$.

10. A *block* in a binary string is a maximal substring consisting of the same symbol. For example, the string 0100011 has four blocks: 0, 1, 000, and 11. Let $H(n)$ denote the number of binary strings of length n that have no odd length blocks of 1's. For example, $H(4) = 5$:

$$0000 \quad 1100 \quad 0110 \quad 0011 \quad 1111$$

    Develop a recursive definition for $H(n)$, and justify why it is correct. Then find a closed form for H using repeated substitution.

11. Consider the following recursively defined function.

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 4T\left(\frac{n}{2}\right) + \log_2 n, & \text{otherwise} \end{cases}$$

    Use repeated substitution to come up with a closed form expression for $T(n)$, *when* $n = 2^k$; i.e., n is a power of 2. You will need to use the following identity:

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

12. Analyse the worst-case runtime of `fast_rec_mult`.
13. Analyse the runtime of each of the following recursive algorithms. It's up to you to decide whether you should use repeated substitution or the Master Theorem to find the asymptotic bound.

(a)

```
def sum(A):
  if len(A) == 0:
    return 1
  else:
    return A[0] + sum(A[1..len(A)-1])
```

(b)

```
def fun(A):
  if len(A) < 2:
    return len(A) == 0
  else:
    return fun(A[2..len(A)-1])
```

(c)

```
def double_fun(A):
  n = len(A)
  if n < 2:
    return n
  else:
    return double_fun(A[0..n-2]) + double_fun(A[1..n-1])
```

(d)

```
def mystery(A):
  if len(A) <= 1:
    return 1
  else:
    d = len(A) // 4
    s = mystery(A[0..d-1])
    i = d
    while i < 3 * d:
      s += A[i]
      i += 1
```

```
    s += mystery(A[3*d..len(A)-1])
    return s
```

14. Recall the recurrence for the worst-case runtime of quicksort:

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ T(|L|) + T(|G|) + dn, & \text{if } n > 1 \end{cases}$$

where L and G are the partitions of the list. Clearly, how the list is partitioned matters a great deal for the runtime of quicksort.

(a) Suppose the lists are always evenly split; that is, $|L| = |G| = \frac{n}{2}$ at each recursive call. Find a tight asymptotic bound on the runtime of quicksort using this assumption.

For simplicity, we'll ignore the fact that each list really would have size $\frac{n-1}{2}$.

(b) Now suppose that the lists are always very unevenly split: $|L| = n - 2$ and $|G| = 1$ at each recursive call. Find a tight asymptotic bound on the runtime of quicksort using this assumption.

Program Correctness

In our study of algorithms so far, we have mainly been concerned with their worst-case running time. While this is an important consideration of any program, there is arguably a much larger one: program **correctness**! That is, while it is important for our algorithms to run quickly, it is more important that they work! You are used to testing your programs to demonstrate their correctness, but your confidence depends on the quality of your testing.

Frankly, developing high-quality tests takes a huge amount of time – much longer than you probably spent on it in CSC148!

In this chapter, we'll discuss methods of formally *proving* program correctness, without writing any tests at all. We cannot overstate the importance of this technique: a test suite cannot possibly test a program on all possible inputs (unless it is a *very* restricted program), and so a proof is the only way we can ensure that our programs are actually correct on *all* inputs. Even for larger software systems, which are far too complex to formally prove their correctness, the skills you will learn in this chapter will enable you to *reason* more effectively about your code; essentially, what we will teach you is the art of semantically tracing through code.

By "semantically" we mean your ability to derive meaning from code, i.e., identify exactly what the program does. This contrasts with program *syntax*, things like punctuation and (in Python) indentation.

What is Correctness?

You may be familiar with the most common tools used to specify program correctness: *preconditions* and *postconditions*. Formally, a precondition of a function is a property that an input to the function *must* satisfy in order to guarantee that the function will work properly. A postcondition of a function is a property that *must* be satisfied after the function completes. Most commonly, this refers to properties of a return value, though it can also refer to changes to the variables passed in as with the implementation of quicksort from the previous chapter (which didn't return anything but instead changed the input list `A`).

precondition/postcondition

As a program designer, it is up to you to specify preconditions. This often balances the desire of *flexibility* (allowing a broad range of inputs/usages) with *feasibility* (how much code you want or are able to write).

A single function can have several pre- and postconditions.

Example. Consider the following code for calculating the greatest common divisor of two natural numbers. Its pre- and postconditions are shown.

```
1  def gcd_rec(a, b):
2    '''
3    Pre: a and b are positive integers, and a >= b
4    Post: returns the greatest common divisor of a and b
5    '''
6    if a == 1 or b == 1:
7      return 1
8    else if a mod b == 0:
9      return b
10   else:
11     return gcd_rec(b, a mod b)
```

We'll use `mod` rather than `%` in our code, for clarity.

So preconditions tell us what must be true before the program starts, and postconditions tell us what must be true after the program terminates (assuming it ends at all). We have the following formal statement of correctness. Though it is written in more formal language, note that it really captures what we mean when we say that a program is "correct."

Definition (Program Correctness)**.** Let `f` be a function with a set of preconditions and postconditions. Then `f` is *correct* (with respect to the pre- and postconditions) if the following holds:

program correctness

> For every input `I` to `f`, if `I` satisfies the preconditions, then `f(I)` terminates, and all the postconditions hold after it terminates.

Correctness of Recursive Programs

Consider the code for `gcd_rec` shown in the previous example. Here is its statement of correctness:

> For all $a, b \in \mathbb{Z}^+$ such that $a \geq b$, `gcd_rec(a,b)` terminates and returns $\gcd(a, b)$.

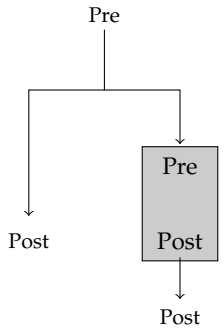
How do we prove that `gcd_rec` is correct? It should come as no surprise that we use *induction*, since you know by now that induction and recursion are closely related concepts. At this point in the course, you should be extremely comfortable with the following informal reasoning: "If a and b are 'small' then we can prove directly using the base case of `gcd_rec` that it is correct. Otherwise, the recursive call happens on 'smaller' inputs, and *by induction*, the recursive call is correct, and hence because of some {math, logic, etc.}, the program also returns the correct value."

Writing full induction proofs that formalise the above logic is tedious, so instead we use the fundamental idea in a looser template. For **each program path** from the first line to a `return` statement, we show that it terminates and that, when it does, the postconditions are satisfied. We do this as follows:

- If the path contains no recursive calls or loops, analyse the code line by line until the `return` statement.
- For each recursive call on the path (if there are any), argue why the preconditions are satisfied at the time of the recursive call, *and* that the recursive call occurs on a "smaller" input than the original call. Then you may assume that the postconditions for the recursive call are satisfied when the recursive call terminates.

  Finally, argue from the last recursive call to the end of the function why the postconditions *of the original function call* will hold.
- For each loop, use a "loop invariant." We will deal with this in the next section.

There is some ambiguity around what is meant by "smaller." We will discuss this shortly.



So in essence, our "predicate" would be "for all inputs `I` of size n that satisfy the precondition of `f`, `f` is correct on `I`.

We have described the *content* of an induction proof; by focusing on tracing values in the code, we are isolating the most important thinking involved. But recall that the core of induction is proving a predicate for all *natural numbers*; thus the final ingredient we need to account for is associating each input with a natural number: its "size." Usually, this will be very simple: the length of a list, or the value of an input number. Let us illustrate this technique using `gcd_rec` as our example.

Example. We will show that `gcd_rec` is correct. There are **three** program paths (this is easy to see, because of the `if` statements). Let's look at each one separately.

- Path 1: the program terminates at line 7. If the program goes into this block, then $a = 1$ or $b = 1$. But in these cases, $\gcd(a, b) = 1$,

because $\gcd(x, 1) = 1$ for all x. Then the postcondition holds, since at line 7 the program returns 1.

- Path 2: the program terminates at line 9. If the program goes into this block, b divides a. Since b is the greatest possible divisor of itself, this means that $\gcd(a, b) = b$, and b is returned at line 9.
- Path 3: the program terminates at line 11. We need to check that the recursive call satisfies its preconditions and is called on a smaller instance. Note that b and $(a \mod b)$ are both at least 1, and $(a \mod b) < b$, so the preconditions are satisfied. Since $a + b > (a \mod b) + b$, the *sum of the inputs* decreases, and so the recursive call is made on a smaller instance.

  Therefore when the call completes, it returns $\gcd(b, a \mod b)$. Now we use the identity that $\gcd(a, b) = \gcd(b, a \mod b)$ to conclude that the original call returns the correct answer.

If you recall the example of using complete induction on ordered pairs, taking the sum of the two components was the size measure we used there, too.

Example. We now look at a recursive example on lists. Here we consider a *randomized* binary search – this is worse than regular binary search in practice, but is useful for our purposes because it shows that the correctness of binary search doesn't depend on the size of the recursive calls.

How is it similar to quicksort?

```
1  def rand_bin_search(A, x):
2    '''
3    Pre: A is a sorted list of numbers, and x is a number
4    Post: Returns true if and only if x is an element of A
5    '''
6    if len(A) == 0:
7      return false
8    else if len(A) == 1:
9      return A[0] == x
10   else:
11     guess = a random number from 0 to len(A) - 1, inclusive
12     if A[guess] == x:
13       return true
14     else if A[guess] > x:
15       return rand_bin_search(A[0..guess-1], x)
16     else:
17       return rand_bin_search(A[guess+1..len(A)-1], x)
```

Proof of correctness. Here there are five different program paths. We'll check three of them, and leave the others as an exercise:

- Path 1: the program terminates at line 7. This happens when `A` is empty; if this happens, `x` is certainly not in `A`, so the program returns the correct value (`false`).
- Path 2: the program terminates at line 9. We compare `A[0]` to `x`, returning `true` if they are equal, and `false` otherwise. Note that if they aren't equal, then `x` cannot be in `A`, since `A` has only one element (its length is one).
- Path 4: the program terminates at line 15. This happens when `len(A) > 1`, and `A[guess] > x`. Because `A` is sorted, and `A[guess] > x`, for every index $i \geq$ `guess`, `A[i] > x`. Therefore the only way `x` could appear in `A` is if it appeared at an index smaller than `guess`.

  Now, let us handle the recursive call. Since `guess` $\leq len(A) - 1$, we have that `guess` $- 1 \leq len(A) - 2$, and so the length of the list in the recursive call is at most $len(A) - 1$; so the recursive call happens on a smaller instance. Therefore, when the recursive call returns, the postcondition is satisfied: it returns `true` if and only if `x` appears in `A[0..guess-1]`. The original function call then returns this value; by the discussion in the previous paragraph, this is the correct value to return, so the postcondition is satisfied.

■

Iterative Programs

In this section, we'll discuss how to handle *loops* in our code. So far, we have been able to determine the exact sequence of steps in each program path (e.g., "Lines 1, 2, 4, and 6 execute, and then the program returns"). However, this is *not* the case when we are presented with a loop, because the sequence of steps depends on the number of times the loop iterates, which in turn depends on the input (e.g., the length of an input list). Thus our argument for correctness cannot possibly go step by step!

We've treated recursive calls as "black boxes" that behave nicely (i.e., can be treated as a single step) as long as their preconditions are satisfied and they are called on smaller inputs.

Instead, we treat the entire loop as a single unit, and give a correctness argument specifically for it separately. But what do we mean for a loop to be "correct"? Consider the following function.

```
1  def avg(A):
2    '''
3    Pre: A is a non-empty list of numbers
4    Post: Returns the average of the numbers in A
5    '''
```

```
6   sum = 0
7   i = 0
8   while i < len(A):
9     sum += A[i]
10    i += 1
11  return sum / len(A)
```

This is the sort of program you wrote in CSC108. Intuitively, it is certainly correct — why? The "hard" work is done by the loop, which calculates the sum of the elements in `A`. The key is to *prove* that this is what the loop does. Coming out of first-year programming, most of you would be comfortable saying that the variable `sum` starts with value 0 before the loop, and after the loop ends it contains the value of the sum of all elements in `A`. But what can we say about the value of `sum` *while the loop is running*?

Clearly, the loop calculates the sum of the elements in `A` one element at a time. After some thought, we determine that the variable `sum` starts with value 0 and in the loop takes on the values `A[0]`, then `A[0] + A[1]`, then `A[0] + A[1] + A[2]`, etc. We formalize this by defining a **loop invariant** for this loop. A loop invariant is a predicate that is true every time the loop-condition is checked (including the check that terminates the loop). Usually, the predicate will depend on which iteration the loop is on, or more generally, the *value(s) of the program variable(s) associated with the loop.* For example, in `avg`, the loop invariant corresponding to our previous intuition is

loop invariant

$$P(i, sum) : sum = \sum_{k=0}^{i-1} A[k]$$

By convention, the empty sum $\sum_{k=0}^{-1} A[k]$ evaluates to 0.

The i and sum in the predicate really correspond to the values of those variables in the code for `avg`. That is, this predicate is stating a *property* of these variables in the code.

Unfortunately, this loop invariant isn't quite right; what if $i > len(A)$? Then the sum is not well-defined, since, for example, `A[len(A)]` is undefined. This can be solved with a common technique for loop invariants: putting bounds on "loop counter" variables, as follows:

$$Inv(i, sum) : 0 \leq i \leq len(A) \wedge sum = \sum_{k=0}^{i-1} A[k].$$

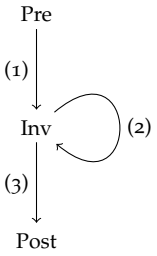
This ensures that the sum is always well-defined, and has the added benefit of explicitly defining a possible range of values on i.

In general, the fewer possible values a variable takes on, the fewer cases you have to worry about in your code.

A loop invariant is *correct* if it is always true at the beginning of every loop iteration, *including the loop check that fails, causing the loop to terminate.* This is why we allowed $i \leq len(A)$ rather than just $i < len(A)$ in the invariant.

How do we prove that loop invariants are correct? The argument is yet another application of induction:

- First, we argue that the loop invariant is satisfied when the loop is reached. (This is arrow (1) in the diagram)
- Then, we argue that *if* the loop invariant is satisfied at the beginning of an iteration, *then* after the loop body executes once (i.e., one loop iteration occurs), the loop invariant still holds. (Arrow (2))
- Finally, after proving that the loop invariant is correct, we show that if the invariant holds when the loop ends, then the postcondition will be satisfied when the program returns. (Arrow (3))



Though this is basically an inductive proof, as was the case for recursive programs, we won't hold to the formal induction structure here.

If we wanted to be precise, we would do induction on the *number of loop iterations* executed.

Example. Let us formally prove that `avg` is correct. The main portion of the proof will be the proof that our loop invariant $Inv(i, sum)$ is correct.

Proof. When the program first reaches the loop, $i = 0$ and $sum = 0$. Plugging this into the predicate yields

$$Inv(0,0): \quad 0 \leq 0 \leq len(A) \wedge 0 = \sum_{k=0}^{-1} A[k],$$

which is true (recall the note about the empty sum from earlier).

Now suppose the loop invariant holds when $i = i_0$, at the beginning of a loop iteration. Let sum_0 be the value of the variable `sum` at this time. The loop invariant we are *assuming* is the following:

$$Inv(i_0, sum_0): \quad 0 \leq i_0 \leq len(A) \wedge sum_0 = \sum_{k=0}^{i_0-1} A[k].$$

What happens next? The obvious answer is "the loop body runs," but this misses one subtle point: if $i_0 = len(A)$ (which is allowed by the loop invariant), the body of the loop doesn't run, and *we don't need to worry about this case*, since we only care about checking what happens to the invariant when the loop actually runs.

Assume that $i_0 < len(A)$, so that the loop body runs. What happens in one iteration? Two things: `sum` increases by $A[i_0]$, and `i` increases by 1. Let sum_1 and i_1 be the values of `sum` and `i` at the end of the loop iteration. We have $sum_1 = sum_0 + A[i_0]$ and $i_1 = i_0 + 1$. Our goal is to prove that the loop invariant holds for i_1 and sum_1, i.e.,

$$Inv(i_1, sum_1): \quad 0 \leq i_1 \leq len(A) \wedge sum_1 = \sum_{k=0}^{i_1 - 1} A[k].$$

Let us check that the loop invariant is still satisfied by sum_1 and i_1. First, $0 \leq i_0 < i_0 + 1 = i_1 \leq len(A)$, where the first inequality came from the loop invariant holding at the beginning of the loop, and the last inequality came from the assumption that $i_0 < len(A)$. The second part of Inv can be checked by a simple calculation:

$$\begin{aligned}
sum_1 &= sum_0 + A[i_0] \\
&= \left(\sum_{k=0}^{i_0 - 1} A[k] \right) + A[i_0] && \text{(by } Inv(i_0, sum_0)\text{)} \\
&= \sum_{k=0}^{i_0} A[k] \\
&= \sum_{k=0}^{i_1 - 1} A[k] && \text{(Since } i_1 = i_0 + 1\text{)}
\end{aligned}$$

Therefore the loop invariant always holds.

The next key idea is that when the loop ends, variable `i` has value $len(A)$, since by the loop invariant it always has value $\leq len(A)$, and if it were strictly less than $len(A)$, another iteration of the loop would run. Then by the loop invariant, the value of `sum` is $\sum_{k=0}^{len(A)-1} A[k]$, i.e., the sum of all the elements in `A`! The final step is to continue tracing until the program returns, which in this case takes just a single step: the program returns `sum / len(A)`. But this is exactly the average of the numbers in A, **because the variable `sum` is equal to their sum**! Therefore the postcondition is satisfied. ■

That might seem like a lot of writing to get to what we said paragraphs ago, but this is a formal argument that *confirms* our intuition.

We also implicitly use here the mathematical definition of "average" as the sum of the numbers divided by how many there are.

Note the deep connection between the loop invariant and the postcondition. There are many other loop invariants we could have tried to prove: for example, $Inv(i, sum): i + sum \geq 0$. But this wouldn't have helped at all in proving the postcondition! When confronting more problems on your own, it will be up to you to determine the right loop invariants for the job. Keep in mind that choosing loop invariants can

usually be done by either taking a high-level approach and mimicking something in the postcondition or taking a low-level approach by carefully tracing through the code on test inputs to try to find patterns in the variable values.

One final warning before we move on: loop invariants describe relationships between variables *at a specific moment in time*. Students often try to use loop invariants to capture *how variables change over time* (e.g., "`i` will increase by `1` when the loop runs"), which creates massive headaches because determining how the code works is the meat of a proof, and shouldn't be shoehorned into a single predicate! When working with more complex code, we take the view that loop invariants are properties that are preserved, even if they don't describe exactly how the code works or exactly what happens! This flexibility is what makes correctness proofs manageable.

Specifically, at the beginning of a particular loop check.

Example. Here we consider a numerical example: a low-level implementation of multiplication.

```
def mult(a,b):
  '''
  Pre: a and b are natural numbers
  Post: returns a * b
  '''
  m = 0
  count = 0
  while count < b:
    m += a
    count += 1
  return m
```

Proof of correctness. The key thing to figure out here is how the loop accomplishes the multiplication. It's clear what it's supposed to do: the variable `m` changes from 0 at the beginning of the loop to `a*b` at the end. How does this change happen? One simple thing we could do is make a table of values of the variables `m` and `count` as the loop progresses:

m	count
0	0
a	1
$2a$	2
$3a$	3
$\vdots$	$\vdots$

Aha: `m` seems to always contain the product of `a` and `count`; and, when the loop ends, `count = b`! This leads directly to the following loop invariant (including the bound on `count`):

$$Inv(m, count) : m = a \times count \wedge count \leq b$$

Consider an execution of the code, with the preconditions satisfied by the inputs. Then when the loop is first encountered, $m = 0$ and $count = 0$, so $m = a \times count$, and $count \leq b$ (since $b \in \mathbb{N}$).

Now suppose the loop invariant holds at the beginning of some iteration, with $m = m_0$ and $count = count_0$. Furthermore, suppose $count_0 < b$, so the loop runs. When the loop runs, `m` increases by `a` and `count` increases by 1. Let m_1 and $count_1$ denote the new values of `m` and `count`; so $m_1 = m_0 + a$ and $count_1 = count_0 + 1$. Since $Inv(m_0, count_0)$ holds, we have

Explicitly, we assume that $Inv(m_0, count_0)$ holds.

$$\begin{aligned} m_1 &= m_0 + a \\ &= a \times count_0 + a && \text{(by invariant)} \\ &= a(count_0 + 1) \\ &= a \times count_1 \end{aligned}$$

Moreover, since we've assumed $count_0 < b$, we have that $count_1 = count_0 + 1 \leq b$. So $Inv(m_1, count_1)$ holds.

Finally, when the loop terminates, we must have $count = b$, since by the loop invariant $count \leq b$, and if $count < b$ another iteration would occur. Then by the loop invariant again, when the loop terminates, $m = ab$, and the function returns m, satisfying the postcondition. ■

Termination

Unfortunately, there is a slight problem with all the correctness proofs we have done so far. We've used phrases like "when the recursive call ends" and "when the loop terminates". But how do we know that the recursive calls and loops end at all? That is, how do we know that a program does not contain an infinite loop or infinite recursion?

This is a serious issue: what beginning programmer *hasn't* been foiled by either of these errors?

The case for recursion is actually already handled by our implicit induction proof structure. Recall that the predicate in our induction proofs is that `f` is correct on inputs of size n; part of the definition of correctness is that the program terminates. Therefore as long as the induction structure holds — i.e., that the recursive calls are getting smaller and smaller — termination comes for free.

The case is a little trickier for loops. As an example, consider the loop in `avg`. Because the loop invariant $Inv(i, sum)$ doesn't say anything about how `i` changes, we can't use it to prove that the loop terminates. But for most loops, including this one, it is "obvious" why they terminate, because they typically have *counter* variables that iterate through a fixed range of values.

The counter role is played by the variable `i`, which goes through the range $\{0, 1, \ldots, len(A)\}$.

This argument would certainly convince us that the `avg` loop terminates, and in general all loops with this form of loop counter terminate. However, not all loops you will see or write will have such an obvious loop counter. Here's an example:

```
1  def collatz(n):
2    ''' Pre: n is a natural number '''
3    curr = n
4    while curr > 1:
5      if curr is even:
6        curr = curr // 2
7      else:
8        curr = 3 * curr + 1
```

In fact, it is an open question in mathematics whether this function halts on all inputs or not. If only we had a computer program that could tell us whether it does!

Therefore we'll now introduce a formal way of proving **loop termination**. Recall that our correctness proofs of recursive functions hinged on the fact that the recursive calls were made on smaller and smaller inputs, until some base case was reached. Our strategy for loops will draw inspiration from this: we associate with the loop a **loop variant** v that has the following two properties:

loop termination

loop variant

(1) v *decreases* with each iteration of the loop

(2) v is always a natural number at the beginning of each loop iteration

If such a v exists, then at some point v won't be able to decrease any further (because 0 is the smallest natural number), and therefore the loop cannot have any more iterations. This is analogous to inputs to recursive calls getting smaller and smaller until a base case is reached.

While this is not the only strategy for proving termination, it turns out to be suitable for most loops involving numbers and lists. When you study more advanced data structures and algorithms, you will discuss more complex arguments for both correctness and termination.

Let us illustrate this technique on our `avg` loop.

Example. *Proof of termination of `avg`.* Even though we've already observed that the loop has a natural loop counter variable i, this variable *increases* with each iteration. Instead, our loop variant will be $v = len(A) - i$. Let us check that v satisfies the properties (1) and (2):

(1) Since at each iteration i increases by 1, and $len(A)$ stays the same, $v = len(A) - i$ decreases by 1 on each iteration.

(2) Note that i and $len(A)$ are both always natural numbers. **But this alone is not enough to conclude that** $v \in \mathbb{N}$; for example, $3, 5 \in \mathbb{N}$ but $(3 - 5) \notin \mathbb{N}$. But the loop invariant we proved included the predicate $0 \leq i \leq len(A)$, and because of this we can conclude that $len(A) - i \geq 0$, so $len(A) - i \in \mathbb{N}$.

This is a major reason we include such loop counter bounds on the loop invariant.

Since we have established that v is a decreasing, bounded variant for the loop, this loop terminates, and therefore `avg` terminates (since every other line of code is a simple step that certainly terminates). ■

Notice that the above termination proof relied on i increasing by 1 on each iteration, and that i never exceeds `len(A)`. That is, we basically just used the fact that `i` was a standard loop counter. Here is a more complex example where there is no obvious loop counter.

Example. Prove that the following function terminates:

```
1  def term_ex(x,y):
2    ''' Pre: x and y are natural numbers. '''
3    a = x
4    b = y
5    while a > 0 or b > 0:
6      if a > 0:
7        a -= 1
8      else:
9        b -= 1
10   return x * y
```

Proof. Intuitively, the loop terminates because when the loop runs, *either* a or b decreases, and will stop when a and b reach 0. To make this argument formal, we need the following loop invariant: $a, b \geq 0$, whose proof we leave as an exercise.

The *loop variant* we define is $v = a + b$. Let us prove the necessary properties for v:

- In the loop, either a decreases by 1, or b decreases by 1. In either case, $v = a + b$ decreases by 1. Therefore v is decreasing.
- Since our loop invariant says that $a, b \geq 0$, we have that $v \geq 0$ as well. Therefore $v \in \mathbb{N}$. ■

Exercises

1. Here is some code that recursively determines the smallest element of a list. Give pre- and postconditions for this function, then prove it is correct according to your specifications.

```
def recmin(A):
  if len(A) == 1:
    return A[0]
  else:
    m = len(A) // 2
    min1 = recmin(A[0..m-1])
    min2 = recmin(A[m..len(A)-1])
    return min(min1, min2)
```

2. Prove that the following code is correct, according to its specifications.

```
def sort_colours(A):
  '''
  Pre: A is a list whose elements are either 'red' or 'blue'
  Post: All red elements in A appear before all blue ones
  '''
  i = 0
  j = 0
  while i < len(A):
    if A[i] is red:
      swap A[i], A[j]
      j += 1
    i +=1
```

3. Prove the following loop invariant for the loop in `term_ex`: $Inv(a, b)$: $a, b \geq 0$.

4. Consider the following modification of the `term_exexample`.

```
def term_ex_2(x,y):
  ''' Pre: x and y are natural numbers '''
  a = x
  b = y
  while a >= 0 or b >= 0:
    if a > 0:
      a -= 1
    else:
      b -= 1
  return x * y
```

   (a) Demonstrate via example that this doesn't always terminate.

   (b) Show why the proof of termination given for `term_ex` fails.

5. For each of the following, state pre- and postconditions that capture what the program is designed to do, then prove that it is correct according to your specifications.

   Don't forget to prove termination (even though this is pretty simple). It's easy to forget about this if you aren't paying attention.

   (a)

```
def mod(n, d):
  r = n
  while r >= d:
    r -= d
  return r
```

   (b)

```
def div(n, d):
  r = n
  q = 0
  while r >= d:
    r -= d
    q += 1
  return q
```

(c)

```
def lcm(a,b):
  x = a
  y = b
  while x != y:
    if x < y:
      x += a
    else:
      y += b
  return x
```

(d)

```
def div3(s):
  r = 0
  t = 1
  i = 0
  while i < len(s):
    r += t * s[i]
    t *= -1
    i += 1
  return r mod 3 == 0
```

(e)

```
def count_zeroes(L):
  z = 0
  i = 0
  while i < len(L):
    if L[i] == 0:
      z += 1
    i +=1
  return z
```

(f)

```
def f(n):
  r = 2
  i = n
  while i > 0:
    r = 3*r -2
```

```
    i -= 1
  return r
```

6. Consider the following code.

```
def f(x):
  ''' Pre: x is a natural number '''
  a = x
  y = 10
  while a > 0:
    a -= y
    y -= 1
  return a * y
```

(a) Give a loop invariant that characterizes the values of a and y.

(b) Show that sometimes this code fails to terminate.

7. In this question, we study two different algorithms for exponentiation: a recursive and iterative algorithm. First, state pre- and postconditions that the algorithms must satisfy (they're the same for the two). Then, prove that each algorithm is correct according to the specifications.

(a)

```
def exp_rec(a, b):
  if b == 0:
    return 1
  else if b mod 2 == 0:
    x = exp_rec(a, b / 2)
    return x * x
  else:
    x = exp_rec(a, (b - 1)/2)
    return x * x * a
```

(b)

```
def exp_iter(a, b):
  ans = 1
  mult = a
  exp = b
```

```
  while exp > 0:
    if exp mod 2 == 1:
      ans *= mult
    mult = mult * mult
    exp = exp // 2
  return ans
```

8. Prove that the following function is correct. Warning: this one is probably the most difficult of these exercises. But, it runs in linear time – pretty amazing!

```
def majority(A):
  '''
  Pre: A is a list with more than half its entries equal to x
  Post: Returns the majority element x
  '''
  c = 1
  m = A[0]
  i = 1
  while i <= len(A) - 1:
    if c == 0:
      m = A[i]
      c == 1
    else if A[i] == m:
      c += 1
    else:
      c -= 1
    i += 1
  return m
```

9. Here we study yet another sorting algorithm, `bubblesort`.

```
def bubblesort(L):
  '''
  Pre: L is a list of numbers
  Post: L is sorted
  '''
  k = 0
  while k < len(L):
    i = 0
    while i < len(L) - k:
      if L[i] > L[i+1]:
        swap L[i] and L[i+1]
      i += 1
    k +=1
```

(a) State and prove an invariant for the inner loop.

(b) State and prove an invariant for the outer loop.

(c) Prove that `bubblesort` is correct, according to its specifications.

10. Consider the following generalization of the `min` function.

```
def extract(A, k):
  pivot = A[0]
  # Use partition from quicksort
  L, G = partition(A[1..len(A) - 1], pivot)
  if len(L) == k - 1:
    return pivot
  else if len(L) >= k:
    return extract(L, k)
  else:
    return extract(G, k - len(L) - 1)
```

(a) Prove that this algorithm is correct.

(b) Analyse the worst-case running time of this algorithm. Hint: this algorithm is known as *quickselect*, and is pretty obviously related to quicksort.

Regular Languages & Finite Automata

In this final chapter, we turn our attention to the study of *finite automata*, a simple model of computation with surprisingly deep applications ranging from vending machines to neurological systems. We focus on one particular application: matching *regular languages*, which are the foundation of natural language processing, including text searching and parsing. This application alone makes automata an invaluable computational tool, one with which you are probably already familiar in the guise of *regular expressions*.

Definitions

We open with some definitions related to strings. An **alphabet** Σ is a *finite* set of symbols, e.g., $\{0,1\}$, $\{a,b,\ldots,z\}$, or $\{0,1,\ldots,9,+,-,\times,\div\}$. A **string** over an alphabet Σ is a *finite* sequence of symbols from Σ. Therefore "0110" is a string over $\{0,1\}$, and "*abba*" and "*cdbaaaa*" are strings over $\{a,b,c,d\}$. The *empty string* "", denoted by ϵ, consists of a sequence of zero symbols from the alphabet. We use the notation Σ^* to denote the set of all strings over the alphabet Σ.

alphabet

string

Σ: Greek letter "Sigma", ϵ: "epsilon"

The **length** of a string $w \in \Sigma^*$ is the number of symbols appearing in the string, and is denoted $|w|$. For example, $|\epsilon| = 0$, $|aab| = 3$, and $|11101010101| = 11$. We use Σ^n to denote the set of strings over Σ of length n. For example, if $\Sigma = \{0,1\}$, then $\Sigma^0 = \{\epsilon\}$ and $\Sigma^2 = \{00,01,10,11\}$.

length

So $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

A **language** L over an alphabet Σ is a subset of strings over Σ: $L \subseteq \Sigma^*$. Unlike alphabets and strings, languages may be either finite or infinite. Here are some examples of languages over the alphabet

language

The "English language" is a subset of the strings that can be formed by all possible combinations of the usual 26 letters.

$\{a, b, c\}$:

$\{\epsilon, a, b, ccc\}$
$\{w \in \{a, b, c\}^* \mid |w| \leq 3\}$
$\{w \in \{a, b, c\}^* \mid w \text{ has the same number of } a\text{'s and } c\text{'s}\}$
$\{w \in \{a, b, c\}^* \mid w \text{ can be found in an English dictionary}\}$

These are pretty mundane examples. Somewhat surprisingly, however, this notion of languages also captures *solutions to computational problems*. Consider the following languages over the alphabet of all standard ASCII characters.

$L_1 = \{A \mid A \text{ is a string representation of a sorted list of numbers}\}$
$L_2 = \{(A, x) \mid A \text{ is a list of numbers, } x \text{ is the minimum of } A\}$
$L_3 = \{(a, b, c) \mid a, b, c \in \mathbb{N} \text{ and } \gcd(a, b) = c\}$
$L_4 = \{(P, x) \mid P \text{ is a Python program that halts when given input } x\}$

E.g., "$[1, 2, 76]$" $\in L_1$

So we can interpret many computer programs as *deciding membership in a particular language*. For example, a program that decides whether an input string is in L_1 is essentially a program that decides whether an input list is sorted.

Membership in L_4 cannot be computed by any program at all — this is the famous Halting Problem!

Solutions to computational problems are just one face of the coin; what about the programs we create to solve them? One of the great achievements of the early computer scientist Alan Turing was the development of the *Turing Machine*, an abstract model of computation that is just as powerful as the physical computers we use today. Unfortunately, Turing Machines are a far more powerful and complex model than is appropriate for this course. Instead, we will study the simpler computational model of *finite automata*, and the class of languages they compute: *regular languages*.

You will study Turing Machines to your heart's content in CSC363/CSC463.

Regular Languages

Regular languages are the most basic kind of languages, and are derived from rather simple language operations. In particular, we define the following three operations for languages $L, M \subseteq \Sigma^*$:

- **Union**: The *union* of L and M is the language $L \cup M = \{x \in \Sigma^* \mid x \in L \text{ or } x \in M\}$. **union**
- **Concatenation**: The *concatenation* of L and M is the language $LM = \{xy \in \Sigma^* \mid x \in L, y \in M\}$. **concatenation**

- **Kleene Star**: The *(Kleene) star* of L is the language $L^* = \{\epsilon\} \cup \{x \in \Sigma^* \mid \exists w_1, w_2, \ldots, w_n \in L \text{ such that } x = w_1 w_2 \ldots w_n, \text{ for some } n\}$. That is, L^* contains the strings that can be broken down into 0 or more smaller strings, each of which is in L.

Kleene star

The star operation can be thought of in terms of union and concatenation as $L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \cdots$

Example. Consider the languages $L = \{a, bb\}$ and $M = \{a, c\}$. Then we have

$$\begin{aligned} L \cup M &= \{a, bb, c\} \\ LM &= \{aa, ac, bba, bbc\} \\ L^* &= \{\epsilon, a, aa, bb, aaa, abb, bba, \ldots\} \\ M^* &= \{\epsilon, a, c, aa, ac, ca, cc, aaa, aac, \ldots\} \end{aligned}$$

Note that $M^* = \{a, c\}^*$ is exactly the strings made up of only a's and c's. This explains the notation Σ^* to denote the set of all strings over the alphabet Σ.

We can now give the following recursive definition of regular languages:

Definition (Regular Language)**.** The set of *regular languages* over an alphabet Σ is defined recursively as follows:

regular language

- $\emptyset$, the empty set, is a regular language.
- $\{\epsilon\}$, the language consisting of only the empty string, is a regular language.
- For any symbol $a \in \Sigma$, $\{a\}$ is a regular language.
- If L, M are regular languages, then so are $L \cup M$, LM, and L^*.

Students often confuse the notation $\emptyset$, ϵ, and $\{\epsilon\}$. First, ϵ is a *string*, while $\emptyset$ and $\{\epsilon\}$ are *sets* of strings. The set $\emptyset$ contains *no* strings (and so has size 0), while $\{\epsilon\}$ contains the single string ϵ (and so has size 1).

Regular languages are sets of strings, and are often infinite. As humans, we are able to leverage our language processing and logical abilities to represent languages; for example, "strings that start and end with the same character" and "strings that have an even number of zeroes" are both simple descriptions of regular languages. What about computers? We could certainly write simple programs that compute either of the above languages, but we have grander ambitions. Specifically, we would like a simple, computer-friendly representation of regular languages so that we could input an *arbitrary* regular language and a string, and the computer would determine whether the string is in the language or not.

Exercise: write these programs.

This is precisely the idea behind the *regular expression (regex)*, a pattern-based string representation of a regular language. Given a regular expression r, we use $\mathcal{L}(r)$ to denote the language *matched* (represented) by r. Here are the elements of regular expressions:

regular expression

Note the almost identical structure to the definition of regular languages themselves.

- $\emptyset$ is a regex, with $\mathcal{L}(\emptyset) = \emptyset$ (matches no string)
- ϵ is a regex, with $\mathcal{L}(\epsilon) = \{\epsilon\}$ (matches only the empty string)
- For all symbols $a \in \Sigma$, a is a regex with $\mathcal{L}(a) = \{a\}$ (matches only the single string a)
- Let r_1, r_2 be regexes. Then $r_1 + r_2$, $r_1 r_2$, and r_1^* are regexes, with $\mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$, $\mathcal{L}(r_1 r_2) = \mathcal{L}(r_1)\mathcal{L}(r_2)$, and $\mathcal{L}(r_1^*) = (\mathcal{L}(r_1))^*$ (matches union, concatenation, and star, respectively)

We follow the unfortunate overloading of symbols of previous course instructors. When we write, for example, $\mathcal{L}(\emptyset) = \emptyset$, the first $\emptyset$ is a *string* with the single character $\emptyset$, which is a regular expression; the second $\emptyset$ is the standard mathematical notation representing the empty set.

It's an easy exercise to prove by structural induction that *every regular language can be matched by a regular expression, and every regular expression matches a language that is regular.* Another way to put it is that a language L is regular if and only if there is a regular expression r such that $L = \mathcal{L}(r)$.

Example. Let $\Sigma = \{0, 1\}$. Describe the language of the regex $01 + 1(0 + 1)^*$. To interpret this regex, we need to understand precedence rules. By convention, these are identical to the standard arithmetic precedence; thus *star has the highest precedence, followed by concatenation, and finally union.* Therefore the complete bracketing of this regex is $(01) + (1((0 + 1)^*))$, but with these precedence rules in place, we only need the brackets around the $(0 + 1)$.

So star is like power, concatenation is like multiplication, and union is like addition.

Let us proceed part by part. The "01" component matches the string 01. The $(0 + 1)^*$ matches all binary strings, because it contains all strings resulting from adding a 0 or 1 at each step. This means that $1(0 + 1)^*$ matches a 1, followed by any binary string. Finally, we take the union of these two: $\mathcal{L}(01 + 1(0 + 1)^*)$ is the set of strings that are either 01 or start with a 1.

Example. Let's go the other way, and develop a regular expression given a description of the following regular language:

$$L = \{w \in \{a, b\}^* \mid w \text{ has length at most } 2\}.$$

Solution:

Note that L is finite, so in fact we can simply list out all of the strings in our regex:

$$\epsilon + a + b + aa + ab + ba + bb$$

Another strategy is to divide up the regex into cases depending on length:

$$\epsilon + (a + b) + (a + b)(a + b)$$

The three parts capture the strings of length 0, 1, and 2, respectively. A final representation would be to say that we'll match two characters,

each of which could be empty, a, or b:

$$(\epsilon + a + b)(\epsilon + a + b).$$

All three regexes we gave are correct! In general, there is more than one regular expression that matches any given regular language.

Example. A regular expression matching the language $L = \{w \in \{0,1\}^* \mid w \text{ has } 11 \text{ as a substring}\}$ is rather straightforward:

$$(0+1)^*11(0+1)^*.$$

But what about the *complement* of L, i.e., the language $\bar{L} = \{w \in \{0,1\}^* \mid w \text{ does not have } 11 \text{ as a substring}\}$? It is more difficult to find a regex for this language because regular expressions specify patterns that should be *m*atched, not avoided. Let's approach this problem by interpreting the definition as "every 1 must be preceded by a 0."

Here is our first attempt:

$$(00^*1)^*.$$

Inside the brackets, 00^*1 matches a non-empty block of 0's followed by a 1, and this certainly ensures that there are no consecutive 1's. Unfortunately, this regular expression matches only a *subset* of $\bar{L}$, and not $\bar{L}$ entirely. For example, the string 1001 is in $\bar{L}$, but can't be matched by this regular expression because the first 1 isn't preceded by any 0's. We can fix this by saying that the first character could possibly be a 1:

$$(\epsilon + 1)(00^*1)^*.$$

There is one last problem: this fails to match any string that ends with 0's, e.g., 0100. We can apply a similar fix as the previous one to allow a block of 0's to be matched at the end:

$$(\epsilon + 1)(00^*1)^*0^*.$$

Some interesting meta-remarks arise naturally from the last example. One is a rather straightforward way of showing that a language and regex are *inequivalent*, by simply producing a string that is in one but not the other. On the other hand, convincing ourselves that a regular expression *correctly* matches a language can be quite difficult; how did we *know* that $(\epsilon + 1)(00^*1)^*0^*$ correctly matches $\bar{L}$? Proving that a regex matches a particular language L is much harder, because we need to show that *every* string in L is matched by the regex, and *every* string not in L is not.

Keep this in mind when you and your friends are arguing about whose regular expressions are correct.

Note that this is a universally quantified statement, while its negation (regex doesn't match L) is existentially quantified. This explains the difference in difficulty.

Our intuition only takes us so far — it is precisely this gap between our gut and true understanding that *proofs* were created to fill. This is,

however, just a little beyond the scope of the course; you have all the necessary ingredients — surprise: induction — but the arguments for all but the simplest languages are quite involved.

We will see later that reasoning about the correctness of deterministic finite automata is just as powerful, and a little simpler.

Example. We finish off this section with a few *corner cases* to illustrate some of the subtleties in our definition, and extend the arithmetic metaphor we hinted at earlier. First, the following equalities show that $\emptyset$ plays the role of the "zero" in regular expressions. Let r be an arbitrary regular expression.

$$\emptyset + r = r + \emptyset = r$$
$$\emptyset r = r\emptyset = \emptyset$$

By equality between regular expressions we mean that they match the same language. That is, $r_1 = r_2 \Leftrightarrow \mathcal{L}(r_1) = \mathcal{L}(r_2)$.

The first is obvious because taking the union of any set with the empty set doesn't change the set. What about concatenation? Recall that concatenation of languages involves taking combinations of strings from the first language and strings from the second; if one of the two languages is empty, then no combinations are possible.

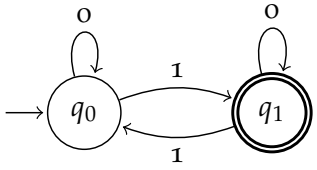
We can use similar arguments to show that ϵ plays the role of the "one":

$$\epsilon r = r\epsilon = r$$
$$\epsilon^* = \epsilon$$

A Suggestive Flowchart

You might be wondering how computers actually match strings to regular expressions. It turns out that regular languages are rather easy to match because of the following (non-obvious!) property:

> You can determine membership in a regular language by reading symbols of the string one at a time, left to right.



Consider the flowchart-type object shown in the figure on the right. Suppose we start at the state marked q_0. Now consider the string 0110, reading the symbols one at a time, left to right, and following the arrows marked by the symbols we read in. It is not hard to see that we end up at state q_0. On the other hand, if the string is 111, we end up at state q_1. The state q_1 is marked as special by a double-border; we say that a string ending up at q_1 is *accepted*, while a string ending at q_0 is *rejected*.

After some thought you may realise that the accepted strings are exactly the ones with an odd number of 1's. A more suggestive way of saying this is that the *language accepted by* this flowchart is the set of strings with an odd number of 1's.

Deterministic Finite Automata

We now use the notion of "flowcharts" to define a simple model of computation. Each one acts as a simple computer program: starting at a particular point, it receives inputs from the user, updates its internal memory, and when the inputs are finished, it outputs `True` or `False`. The following definition is likely one of the most technical you've encountered to date, in this course and others. Make sure you fully understand the prior example, and try to match it to the formal definition as you read it.

Definition (Deterministic Finite Automaton)**.** A **deterministic finite automaton (DFA)** (denoted $\mathcal{D}$) is a quintuple $\mathcal{D} = (Q, \Sigma, \delta, s, F)$ where the components define the various parts of the automaton:

deterministic finite automaton

- Q is the (finite) set of *states* in $\mathcal{D}$
- Σ is the *alphabet* of symbols used by $\mathcal{D}$
- $\delta : Q \times \Sigma \to Q$ is the *transition function* (represents the "arrows")
- $s \in Q$ is the *initial state* of $\mathcal{D}$
- $F \subseteq Q$ is the set of *accepting (final) states* of $\mathcal{D}$

Example. In the introductory example, the state set is $Q = \{q_0, q_1\}$, the alphabet is $\{0, 1\}$, the initial state is q_0, and the set of final states is $\{q_1\}$. We can represent the transition function as a table of values:

Note that there's only one final state in this example, but in general there may be several final states.

Old State	Symbol	New State
q_0	0	q_0
q_0	1	q_1
q_1	0	q_1
q_1	1	q_0

In general, the number of rows in the transition table is $|Q| \cdot |\Sigma|$; each state must have exactly $|\Sigma|$ transitions leading out of it, each labelled with a unique symbol in Σ.

Before proceeding, make sure you understand each of the following statements about how DFAs work:

- DFAs read strings one symbol at a time, from left to right
- DFAs cannot "go back" and reread previous symbols
- At a particular state, once you have read a symbol there is only one arrow (transition) you can follow
- DFAs have a finite amount of memory, since they have a finite number of states
- Inputs to DFAs can be any length
- Because of these last two points, it is impossible for DFAs to always remember *every symbol* they have read so far

A quick note about notation before proceeding with a few more examples. Technically, δ takes as its second argument a single symbol; e.g., $\delta(q_0, 1) = q_1$ (from the previous example). But we can just as easily extend this definition to arbitrary-length strings in the second argument. For example, we can say $\delta(q_0, 11) = q_0$, $\delta(q_1, 1000111) = q_1$, and $\delta(q_0, \epsilon) = q_0$.

For every state q, $\delta(q, \epsilon) = q$.

With this "extended" transition function, it is very easy to symbolically represent the language $\mathcal{L}(\mathcal{D})$ accepted by the DFA $\mathcal{D}$.

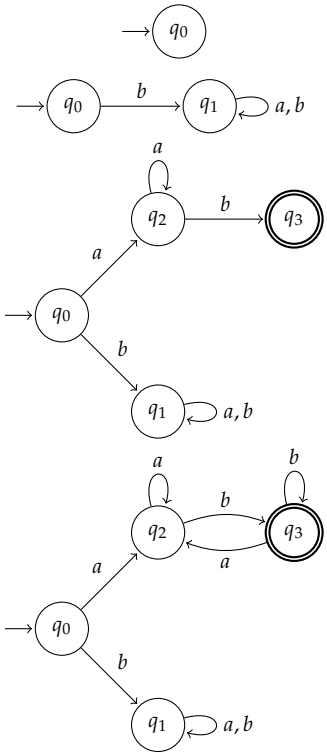
Recall that the language accepted by a DFA is the set of strings accepted by it.

$$\mathcal{L}(\mathcal{D}) = \{w \in \Sigma^* \mid \delta(s, w) \in F\}$$

Example. Let us design a DFA that accepts the following language:

$$L = \{w \in \{a, b\}^* \mid w \text{ starts with } a \text{ and ends with } b\}.$$

Solution:
Consider starting at an initial state q_0.



What happens when you read in an a or a b? Reading in a b should lead to a state q_1 where it's impossible to accept. On the other hand, reading in an a should lead to a state q_2 where we need to "end with a b." The simple way to achieve this is to have q_2 loop on a's, then move to an accepting state q_3 on a b.

What about the transitions for q_3? As long as it's reading b's, it can accept, so it should loop to itself. On the other hand, if it reads an a, it should go back to q_2, and continue reading until it sees another b.

Correctness of DFAs

Like regular expressions, arguing that DFAs are *incorrect* is generally easier than arguing that they are correct. However, because of the rather restricted form DFAs must take, reasoning about their behaviour is a little more amenable than for regular expressions.

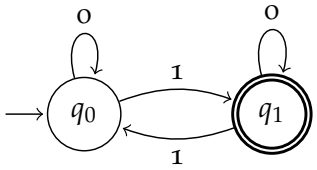
The simple strategy of "pick an arbitrary string in L, and show that it is accepted by the DFA" is hard to accomplish, because the paths taken through the DFA by each accepted string can be quite different; i.e., different strings will probably require substantially different proofs. Therefore we adopt a different strategy. We know that DFAs consist of states and transitions between states; the term *state* suggests that if a string reaches that point, the DFA "knows" something about that string, or it is "expecting" what will come next. We formalize this notion by characterizing *for each state* precisely what must be true about the strings that reach it.

Definition (State invariant)**.** Let $\mathcal{D} = (Q, \Sigma, \delta, s, F)$ be a DFA. Let $q \in Q$ be a state of the DFA. We define a **state invariant** for q as a predicate $P(x)$ (over domain Σ^*) such that for every string $w \in \Sigma^*$, $\delta(s, w) = q$ *if and only if* $P(w)$ is true.

state invariant

Note that the definition of state invariant uses an if and only if. We aren't just giving properties that the strings reaching q must satisfy; we are defining *precisely* which strings reach q. Let us see how state invariants can help us prove that DFAs are correct.

Example. Consider the following language over the alphabet $\{0, 1\}$: $L = \{w \mid w \text{ has an odd number of } 1\text{'s}\}$, and the DFA shown. Prove that the DFA accepts precisely the language L.



Proof. It is fairly intuitive why this DFA is correct: strings with an even number of 1's go to q_0, and transition to q_1 upon reading a 1 (where the string now has an odd number of 1's). Here are some *state invariants* for the two states:

$$\delta(q_0, w) = q_0 \Leftrightarrow w \text{ has an even number of 1's}$$
$$\delta(q_0, w) = q_1 \Leftrightarrow w \text{ has an odd number of 1's}$$

Here are two important properties to keep in mind when designing state invariants:

- The state invariants should be *mutually exclusive*. That is, there should be no overlap between them; no string should satisfy two different state invariants. *Otherwise, to which state would the string go?*
- The state invariants should be *exhaustive*. That is, they should cover all possible cases; every string in Σ^*, including ϵ, should satisfy one of the state invariants. *Otherwise, the string goes nowhere.*

These conditions are definitely satisfied by our two invariants above, since every string has either an even or odd number of 1's.

Next, we want to *prove* that the state invariants are correct. We do this in two steps.

- **Show that the empty string ϵ satisfies the state invariant of the initial state.** In our case, the initial state is q_0; ϵ has zero 1's, which is even; therefore the state invariant is satisfied by ϵ.
- **For each transition $q \xrightarrow{a} r$, show that if a string w satisfies the invariant of state q, then the string wa satisfies the invariant of r.** (That is, each transition respects the invariants.) There are four transitions in our DFA. For the two 0-loops, appending a 0 to a string doesn't change the number of 1's in the string, and hence if w contains an even (odd) number of 1's, then $w0$ contains an even (odd) number of 1's as well, so these two transitions are correct.

  On the other hand, appending a 1 increases the number of 1's in a string by one. So if w contains an even (odd) number of 1's, $w1$ contains an odd (even) number of 1's, so the transitions between q_0 and q_1 labelled 1 preserve the invariants.

The astute reader will note that we are basically doing a proof by induction on the length of the strings. Like our proofs of program correctness, we "hide" the formalities of induction proofs and focus only on the content.

Thus we have proved that the state invariants are correct. The final step is to show that **the state invariants of the accepting state(s) precisely describe the target language**. This is very obvious in this case, because the only accepting state is q_1, and its state invariant is exactly the defining characteristic of the target language L. ■

Remember that in general, there can be more than one accepting state!

Limitations of DFAs

The simplicity of the DFA model enables proofs of correctness, as shown above. This simplicity is also useful for reasoning about the model's limitations. In this section, we'll cover two examples. First, we prove a lower bound on the *number of states* required in a DFA accepting a particular language. Then, we'll show that certain languages cannot be accepted by DFAs of *any* size!

Example. Consider the language

$$L = \{w \in \{0,1\}^* \mid w \text{ has at least three 1's.}\}.$$

We'll prove that any DFA accepting this language has at least 4 states.

Proof. Suppose there exists a DFA $\mathcal{D} = (Q, \Sigma, \delta, s, F)$ that accepts L and has fewer than four states. Consider the four strings $w_0 = \epsilon$, $w_1 = 1$, $w_2 = 11$, and $w_3 = 111$. By the *Pigeonhole Principle*, two of these strings reach the same state in $\mathcal{D}$ from the initial state. Suppose $0 \leq i < j \leq 3$, and $\delta(s, w_i) = \delta(s, w_j) = q$ for some state $q \in Q$. (That is, w_i and w_j both reach the same state q.)

The *Pigeonhole Principle* states that if $m > n$, and you are putting m pigeons into n holes, then two pigeons will go into the same hole.

Now, since w_i and w_j reach the same state, they are *indistinguishable prefixes* to the DFA; this means that any strings of the form $w_i x$ and $w_j x$ will end up at the same state in the DFA, and hence are both accepted or both rejected. However, suppose $x = w_{3-j}$. Then $w_i x = w_{i+3-j}$ and $w_j x = w_{j+3-j} = w_3$. Then $w_i x$ contains $3 - j + i$ 1's, and $w_j x$ contains three 1's. But since $i < j$, $3 - j + i < 3$, so $w_i x \notin L$, while $w_j x \in L$. Therefore $w_i x$ and $w_j x$ cannot end up at the same state in $\mathcal{D}$, a contradiction! ■

The key idea in the above proof was that the four different strings $\epsilon, 1, 11, 111$ all had to reach different states, because there were *suffixes* that could distinguish any pair of them. In general, to prove that a language L requires at least k states in a DFA to accept it, it suffices to give a set of k strings, each of which is distinguishable from the others with respect to L.

These suffixes were the $x = w_{3-j}$ in the proof.

Two strings w_1 and w_2 are "distinguishable with respect to L" if there is a suffix x such that $w_1 x \in L$ and $w_2 x \notin L$, or vice versa.

Now we turn to a harder problem: proving that some languages cannot be accepted by DFAs of *any* size.

Example. Consider the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. Prove that no DFA accepts L.

Proof. We'll prove this by contradiction again. Suppose there is a DFA $\mathcal{D} = (Q, \Sigma, \delta, s, F)$ that accepts L. Let $k = |Q|$, the number of states

in $\mathcal{D}$. Now here is the key idea: $\mathcal{D}$ has only k states of "memory," whereas to accept L, you really have to remember the exact number of 0's you've read in so far, so that you can match that number of 1's later. So we should be able to "overload" the memory of $\mathcal{D}$.

Here's how: consider the string $w = 0^{k+1}$, i.e., the string consisting of $k+1$ 0's. Since there are only k states in $\mathcal{D}$, the path that w takes through $\mathcal{D}$ must involve a loop starting at some state q. That is, we can break up w into three parts: $w = 0^a 0^b 0^c$, where $b \geq 1$, $\delta(s, 0^a) = q$, and $\delta(q, 0^b) = q$.

This loop is dangerous for the DFA! Because reading 0^b causes a loop that begins and ends at q, the DFA forgets whether it has read 0^b or not; thus the strings $0^a 0^c$ and $0^a 0^b 0^c$ reach the same state, and are now indistinguishable to $\mathcal{D}$. But of course these two strings **are** distinguishable with respect to L: $0^a 0^c 1^{a+c} \in L$, but $0^a 0^b 0^c 1^{a+c} \notin L$. ■

The DFA has lost track of the number of 0's.

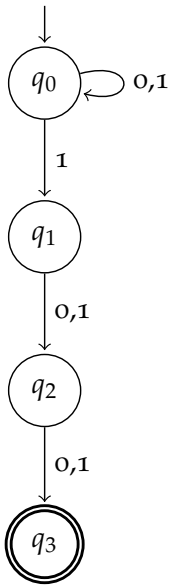
Nondeterminism

Consider the following language over the alphabet $\{0,1\}^*$: $L = \{w \mid$ the third last character of w is 1$\}$. You'll see in the Exercises that a DFA takes at least 8 states to accept L, using the techniques we developed in the previous section. Yet there is a very short regular expression that matches this language: $(0+1)^*1(0+1)(0+1)$. Contrast this with the regular expression $(0+1)(0+1)1(0+1)^*$, matching strings whose third character is 1; this has a simpler 5-state DFA.

You can prove this in the Exercises.

Why is it hard for DFAs to "implement" the former regex, but easy to implement the latter? The fundamental problem is the uncertainty associated with the Kleene star. In the former case, a DFA cannot tell *how many characters to match with the initial* $(0+1)^*$ *segment*, before moving on to the 1! This is not an issue in the latter case, because DFAs read left to right, and so have no problem reading the first three characters, and then matching the rest of the string to the $(0+1)^*$.

On the other hand, consider the automaton to the right. This is not *deterministic* because it has a "choice": reading in a 1 at q_0 can loop back to q_0 or move to q_1. Moreover, reading in a 0 or a 1 at q_3 leads nowhere! But consider this: for any string whose third last character is indeed a 1, there is a "correct path" that leads to the final state q_3. For example, for the string 0101110, the correct path would continuously loop at q_0 for the prefix 0101, then read the next 1, transition to q_1,



then read the remaining 10 to end at q_3, and accept.

BUT WAIT, you say, isn't this like cheating? How did the automaton "know" to loop at the first two 1's, then transition to q_1 on the third 1? We will define our model in this way first, so bear with us. The remarkable fact we'll show later is that, while this model seems more powerful than plain old DFAs, in fact every language that we can accept in this model can also be accepted with a DFA.

Cheating doesn't help.

A **Nondeterministic Finite Automaton (NFA)** is defined in a similar fashion to a DFA: it is a quintuple $\mathcal{N} = (Q, \Sigma, \delta, s, F)$, with Q, Σ, s, and F playing the same roles as before. The *transition function* δ now maps to *sets of states* rather than individual states; that is, $\delta : Q \times \Sigma \to 2^Q$, where 2^Q represents the set of all subsets of Q. For instance, in the previous example, $\delta(q_0, 1) = \{q_0, q_1\}$, and $\delta(q_3, 0) = \emptyset$.

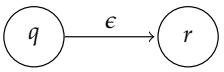
nondeterministic finite automaton

We think of $\delta(q, a)$ here as representing the *set of possible states* reachable from q by reading in the symbol a. This is extended in the natural way to $\delta(q, w)$ for arbitrary length strings w to mean all states reachable from q by reading in the string w. Note that for state q and next symbol a, if $\delta(q, a) = \emptyset$ then this path "aborts," i.e., the NFA does not continue reading more characters for this path.

We say that a string w is *accepted* by NFA $\mathcal{N}$ if $\delta(s, w) \cap F \neq \emptyset$; that is, if *there exists* a final state reachable from the initial state by reading the string w. Note that the existential quantifier used in the definition of acceptance is an integral part of the definition: we don't require that *every* path leading out of s along w reach a final state, only one. This formalizes the notion that it be *possible to choose* the correct path, even from among many rejecting or aborted paths.

On the other hand, if a string is rejected by an NFA, this means all possible paths that string could take either aborted or ended at a rejecting state.

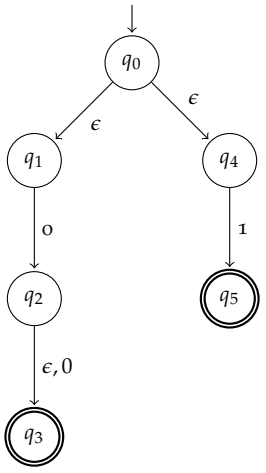
ϵ-transitions



ϵ-transition

We can augment NFAs even further through the use of **ϵ-transitions**. These are nondeterministic transitions that do not require reading in a symbol to activate. That is, if you are currently at state q on an NFA, and there is an ϵ-transition from q to another state r, then you can transition to r without reading the next symbol in the string.

For instance, the NFA on the right accepts the string 0 by taking the ϵ-transition to q_1, reading the 0 to reach q_2, then taking another ϵ-transition to q_3.



As was the case for nondeterminism, ϵ-transitions do not actually

add any power to the model, although they can be useful in certain constructions that we'll use in the next section.

Equivalence of Definitions

So far in this chapter, we have used both DFAs and regular expressions to represent the class of regular languages. We have taken for granted that DFAs are sufficient to represent regular languages; in this section, we will prove this formally. There is also the question of nondeterminism: do NFAs accept a larger class of languages than DFAs? In this section, we'll show that the answer, somewhat surprisingly, is *no*. Specifically, we will sketch a proof of the following theorem.

Theorem (Equivalence of Representations of Regular Languages)**.** *Let L be a language over an alphabet Σ. Then the following are equivalent:*

(1) There is a regular expression that matches L.

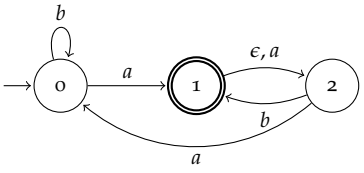
(2) There is a deterministic finite automaton that accepts L.

(3) There is a nondeterministic finite automaton (possibly with ϵ-transitions) that accepts L.

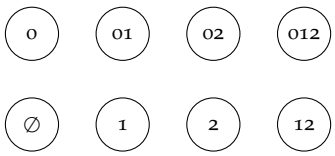
Proof. If you aren't familiar with theorems asserting the equivalence of multiple statements, what we need to prove is that any one of the statements being true implies that all of the others must also be true. We are going to prove this by showing the following *chain of implications*: $(3) \Rightarrow (2) \Rightarrow (1) \Rightarrow (3)$.

We only sketch the main ideas of the proof. For a more formal treatment, see Sections 7.4.2 and 7.6 of Vassos Hadzilacos' course notes.

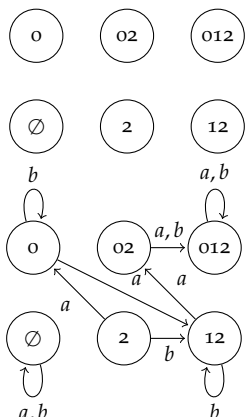
$(3) \Rightarrow (2)$. Given an NFA, we'll show how to construct a DFA that accepts the same language. Here is the high-level idea: nondeterminism allows you to "choose" different paths to take through an automaton. After reading in some characters, the possible path choices can be interpreted as the NFA being simultaneously in some set of states. Therefore we can model the NFA as transitioning between *sets* of states each time a symbol is read. Rather than formally defining this construction, we'll illustrate this on an example NFA (shown right).



There are $2^3 = 8$ possible subsets of states in this NFA; to construct a DFA, we start with one state for each subset (notice that they are labelled according to which states of the NFA they contain, so state 02 corresponds to the subset $\{0, 2\}$).



But, we notice that the ϵ-transition between 1 and 2 ensures that every time we could be in state 1 of the NFA, we could also be in state

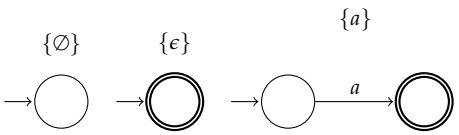


2. Therefore we can ignore states 01 and 1 in our construction, leaving us with just six states.

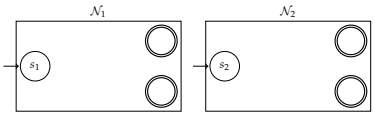
Next, we put in transitions between the states of the DFA. Consider the subset $\{0, 2\}$ of states in the NFA. Upon reading the symbol a, we could end up at all three states, $\{0, 1, 2\}$ (notice that to reach 2, we must transition from 2 to 1 by reading the a, then use the ϵ-transition from 1 back to 2). In our constructed *DFA*, there is a transition between $\{1, 2\}$ and $\{0, 1, 2\}$ on symbol a. So in general, we look at *all possible* outcomes starting from a state in subset S and reading symbol a. Repeating this for all subsets yields the following transitions.

Next, we need to identify initial and final states. The initial state of the NFA is 0, and since there are no ϵ-transitions, the initial state of the constructed DFA is $\{0\}$. The final states of the DFA are exactly the subsets containing the final state 1 of the NFA. Finally, we can simplify the DFA considerably by removing the states $\emptyset$ and $\{2\}$, which cannot be reached from the initial state.

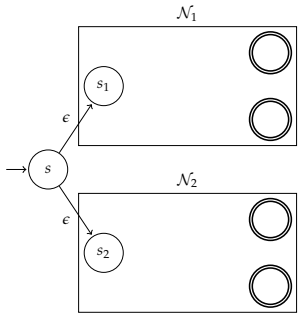
(1) $\Rightarrow$ (3). In this part, we show how to construct NFAs from regular expressions. Note that regular expressions have a recursive definition, so we can actually prove this part using structural induction. First, we show standard NFAs for accepting $\emptyset$, $\{\epsilon\}$, and $\{a\}$ (for a generic symbol a).



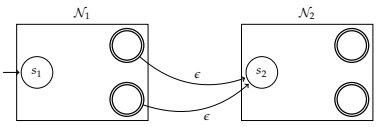
Next, we show how to construct NFAs for union, concatenation, and star, the three recursive operations used to define regular expressions. Note that because we're using structural induction, it suffices to show how to perform these operations on NFAs; that is, given two NFAs $\mathcal{N}_1$ and $\mathcal{N}_2$, construct NFAs accepting $\mathcal{L}(\mathcal{N}_1) \cup \mathcal{L}(\mathcal{N}_2)$, $\mathcal{L}(\mathcal{N}_1)\mathcal{L}(\mathcal{N}_2)$, and $(\mathcal{L}(\mathcal{N}_1))^*$. We use the notation on the right to denote generic NFAs; the two accepting states on the right side of each box symbolize *all* accepting states of the NFAs, and their start states are s_1 and s_2, respectively.



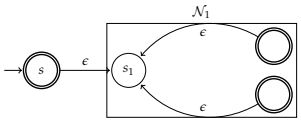
First consider union. This can be accepted by the NFA shown to the right. Essentially, the idea is that starting in a new start state, we "guess" whether the word will be accepted by $\mathcal{N}_1$ or $\mathcal{N}_2$ by ϵ-transitioning to either s_1 or s_2, and then see if the word is actually accepted by running the corresponding NFA.



For concatenation, we start with the first NFA $\mathcal{N}_1$, and then every time we reach a final state, we "guess" that the matched string from $\mathcal{L}(\mathcal{N}_1)$ is complete, and ϵ-transition to the start state of $\mathcal{N}_2$.



Finally, for the Kleene star we perform a similar construction, except that the final states of $\mathcal{N}_1$ ϵ-transition to s_1 rather than s_2. To possibly accept ϵ, we add a new initial state that is also accepting.



(2) $\Rightarrow$ (1). Finally, we show how to construct regular expressions from DFAs. This is the hardest construction to prove, so our sketch here will be especially vague. The key idea is the following:

> For any two states q, r in a DFA, there is a regular expression that matches precisely the strings w such that $\delta(q, w) = r$; i.e., the strings that induce a path from q to r.

Let $\mathcal{D} = (Q, \Sigma, \delta, s, F)$ be a DFA with n states, $Q = \{1, \ldots, n\}$. For each $i, j \in Q$, we define the sets $L_{ij} = \{w \mid \delta(i, w) = j\}$; our ultimate goal is to show that every L_{ij} can be matched by a regular expression. To do this, we use a clever induction argument. For every $0 \leq k \leq n$, let

This part was first proved by Stephen Kleene, the inventor of regular expressions and after whom the Kleene star is named.

$$L_{ij}(k) = \{w \mid \delta(i, w) = j, \text{ and only states } \leq k \text{ are passed between } i \text{ and } j\}$$

Note that $L_{ij}(0)$ is the set of strings where there must be *no* intermediate states, i.e., w is a symbol labelling a transition directly from i to j. Also, $L_{ij}(n) = L_{ij}$: no restrictions are placed on the states that can be passed. We will show how to inductively build up regular expressions matching each of the $L_{ij}(k)$, where the induction is done on k. First, the base case, which we've already described intuitively:

Formally, our predicate is $P(k)$: "For all states i and j, the set $L_{ij}(k)$ can be matched by a regex."

$$L_{ij}(0) = \begin{cases} \{a \in \Sigma \mid \delta(i, a) = j\}, & \text{if } i \neq j \\ \{a \in \Sigma \mid \delta(i, a) = j\} \cup \{\epsilon\}, & \text{if } i = j \end{cases}$$

Note that when $i = j$, we need to include ϵ, as this indicates the trivial act of following no transition at all. Since the $L_{ij}(0)$ are finite sets (of symbols), we can write regular expressions for them (e.g., if $L_{ij}(0) = \{a, c, f\}$, the regex would be $a + c + f$).

You can prove this fact in the Exercises.

Finally, here is the *recursive definition* of the sets that will allow us to construct regular expressions: it defines $L_{ij}(k+1)$ in terms of some $L_{\ldots}(k)$, using only the operations of union, concatenation, and star:

$$L_{ij}(k+1) = L_{ij}(k) \cup \left(L_{i,k+1}(k) L_{k+1,k+1}(k)^* L_{k+1,j}(k)\right)$$

Therefore, given regular expressions for $L_{ij}(k)$, $L_{i,k+1}(k)$, $L_{k+1,k+1}(k)$, and $L_{k+1,j}(k)$, we can construct a regular expression for $L_{ij}(k+1)$.

■

Exercises

1. For each of the following regular languages over the alphabet $\Sigma = \{0, 1\}$, design a regular expression and DFA which accepts that language. For which languages can you design an NFA that is substantially smaller than your DFA?

   (a) $\{w \mid w \text{ contains an odd number of 1's}\}$
   (b) $\{w \mid w \text{ contains exactly two 1's}\}$
   (c) $\{w \mid w \text{ contains 011}\}$
   (d) $\{w \mid w \text{ ends with 00}\}$
   (e) $\{w \mid |w| \text{ is a multiple of 3}\}$
   (f) $\{w \mid$ every 0 in w is eventually followed by a $1\}$
   (g) $\{w \mid w \text{ does not begin with 10}\}$
   (h) $\{w \mid w \text{ is the binary representation of a multiple of three}\}$
   (i) $\{w \mid w \text{ contains both 00 and 11 as substrings}\}$
   (j) $\{0^n 1^m \mid m, n \in \mathbb{N}, m + n \text{ is even}\}$
   (k) $\{w \mid w \text{ contains a substring of length at most 5 that has at least three 1's}\}$
   (l) $\{w \mid w \text{ has an even number of zeroes, or exactly 2 ones}\}$

2. Let $L = \{w \in \{0, 1\}^* \mid$ the third character of w is a $1\}$. Prove that every DFA accepting L has at least 5 states.

3. Let $L = \{w \in \{0, 1\}^* \mid$ the third last character of w is a $1\}$. Prove that every DFA accepting L has at least 8 states. Hint: Consider the 8 binary strings of length 3.

   For a bonus, what is the smallest DFA you can find that accepts L? It will have at least 8 states!

4. Prove by induction that every finite language can be represented by a regular expression. (This shows that all finite languages are regular.)

5. Prove that the following languages are not regular.

   (a) $\{a^{n^2} \mid n \in \mathbb{N}\}$
   (b) $\{xx \mid x \in \{0, 1\}^*\}$
   (c) $\{w \in \{a, b\}^* \mid w \text{ has more } a\text{'s than } b\text{'s}\}$
   (d) $\{w \in \{0, 1\}^* \mid w \text{ has two blocks of 0's with the same length}\}$
   (e) $\{a^n b^m c^{n-m} \mid n \geq m \geq 0\}$

   A block is a *maximal* substring containing the same character; for example, the string 00111000001 has four blocks: 00, 111, 00000, and 1.

6. Recall that the complement of a language $L \subseteq \Sigma^*$ is the set $\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$.

   (a) Given a DFA $\mathcal{D} = (Q, \Sigma, \delta, s, F)$ that accepts a language L, describe how you can construct a DFA that accepts $\overline{L}$.
   (b) Let L be a language over an alphabet Σ. Prove that if L is not regular, then the complement of L is not regular.

7. A regular expression r is *star-free* if it doesn't contain any star operations. Prove that for every star-free regex r, $\mathcal{L}(r)$ is finite.

8. The *suffix operator* $Suff$ takes as input a regular language, and outputs all possible suffixes of strings in the language. For example, if $L = \{aa, baab\}$ then

$$Suff(L) = \{\epsilon, a, aa, b, ab, aab, baab\}.$$

   Prove that if L is a regular language, then so is $Suff(L)$. (Hint: recall the definition of regular languages, and use structural induction!)

9. The *prefix operator* Pre takes as input a regular language, and outputs all possible prefixes of strings in the language. For example, if $L = \{aa, baab\}$ then

$$Pre(L) = \{\epsilon, a, aa, b, ba, baa, baab\}.$$

   Prove that if L is a regular language, then so is $Pre(L)$. (Hint: recall the definition of regular languages, and use structural induction!)

In Which We Say Goodbye

With CSC236 complete, you have now mastered the basic concepts and reasoning techniques vital to your computer science career both at this university and beyond. You have learned how to analyse the efficiency of your programs, both iterative and recursive. You have also learned how to argue formally that they are *correct* by using program specifications (pre- and postconditions) and loop invariants. You studied the finite automaton, a simple model of computation with far-reaching consequences.

Where to from here? Most obviously, you will use your skills in **CSC263** and **CSC373**, where you will study more complex data structures and algorithms. You will see first-hand the real tools with which computers store and compute with large amounts of data, facing real-world problems as ubiquitous as sorting – but whose solutions are not nearly as straightforward. If you were intrigued by the idea of provably correct programs, you may want to check out **CSC410**; if you liked the formal logic you studied in CSC165, and are interested in more of its computer science applications (of which there are many!), **CSC330**, **CSC438**, **CSC465**, and **CSC486** would be good courses to consider. If you'd like to learn about more powerful kinds of automata and more complex languages, **CSC448** is the course for you. Finally, **CSC463** tackles *computability and complexity theory*, the fascinating study of the inherent hardness of problems.

For more information on the above, or other courses, or any other matter academic, professional, or personal, come talk to any one of us in the Department of Computer Science! Our {doors, ears, minds} are always open.

Be the person your dog thinks you are.

Appendix: Runtime Analysis

(Written by Dr. Daniel Zingaro, influenced by discussions with Dr. Cusack and David Liu.)

This appendix introduces three formal notations for analyzing the runtime of algorithms: O, Ω, and Θ. We use these notations in this course to characterize the running times of algorithms in ways that are unrelated to some particular computer, programming language, or operating system.

Here are two Python functions of the sort that you saw in CSC108 and CSC148.

```
def slow(n):
  x = 0
  for i in range(n):
    for j in range(n):
      for k in range(n):
        x = x + 1
  return x
```

```
def faster(n):
  x = 0
  for i in range(n):
    for j in range(n):
      x = x + 1
  return x
```

You'll recall from earlier courses that `faster` is a faster algorithm than `slow`. Why? The intuition is that the triply-nested for-loops of

`slow` are slower than the doubly-nested loops of `faster`. In `slow`, the `x = x + 1` assignment statement happens n^3 times, whereas it happens only n^2 times in `faster`. In this course, we won't be able to rely on such informal ad-hoc analyses, because:

- Some of our functions will be recursive, not iterative, and so there are no loops to count anyway! We want a technique that "works" for recursive functions, too.
- Sometimes, the runtime complexity is not apparent from the loop structure. Instead, we'll want a technique that takes an expression for the number of steps, and returns a bound like n^2 or n^3.

The notations introduced in this Appendix solve both of these problems.

Big O

We'll start with the O notation ("big oh"), which is used to give an upper-bound to an expression.

We say that $f(n)$ is **Big-O** of $g(n)$, written as $f(n) \in O(g(n))$, iff there are positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \text{ for all } n \geq n_0.$$

This definition captures the idea that $g(n)$ is an upper-bound of $f(n)$. It doesn't say anything about being a tight upper-bound, though. For example, it is true that $3n = O(n)$, but it is also true that $3n = O(n^2)$ or $3n = O(2^n)$. Sometimes, people informally use O to mean "tight bound", but this is **not** how it is formally defined.

Note that sometimes you will see $f(n) \in O(g(n))$ written with an equals sign, like $f(n) = O(g(n))$. There's really no equality going on here, so you should continue to think of it as set membership: f is a function in the set of functions that grow no faster than g.

Here's how to think about the two constants n_0 and c:

- n_0 is a point at which g is at least as large as f, and g stays at least as large as f at larger values of $n > n_0$. That is, at $n_0, n_0 + 1, n_0 + 2, \ldots$, g is at least as large as f. This captures the idea that we don't care about the relative sizes of f and g when n is small. We just have to find a point, eventually, where g is at least as large as f.

- If we didn't have the c constant, then we'd be stuck saying stuff like $3n = O(3n)$. But we don't care about the 3 here: what really matters is the n. The constant c is what allows us to "throw away" coefficients and smaller terms, focusing us only on the largest term and ignoring its coefficient.

To prove that $f(n) = O(g(n))$, we start with $f(n)$ and use a chain of $\leq$ inequalities or $=$ equalities to arrive at $cg(n)$ for some $c > 0$. Along the way, we'll have to keep track of some constant n_0 for which the inequalities are true for all values of n at least as large as n_0. Let's see how such a proof goes.

Example. Prove that $100n + 10000 = O(n^2)$.

Proof. Intuitively, you shouldn't be surprised that this is true: n^2 on the right grows more quickly than the n and constant terms on the left. To begin the formal proof, think about how $100n$ compares to $100n^2$. As long as $n \geq 1$, we have that $100n \leq 100n^2$, because $100n^2$ has an extra multiplicative n term that makes the expression larger. So, we've got an upper-bound for $100n$ of $100n^2$, and this is good news, because we're trying to get the left side $100n + 10000$ to be a bunch of n^2 terms.

Let's continue by comparing 10000 to $10000n^2$. Again, when $n \geq 1$, we have that $10000 \leq 10000n^2$. So, we have an upper-bound for 10000 of $10000n^2$. Let's write out what we've got so far, in an equational style:

$$\begin{aligned} 100n + 10000 &\leq 100n^2 + 10000n^2 \\ &= 10100n^2 \end{aligned}$$

We already know our n_0: it's 1. Additionally, from this chain of inequalities, we also get our c: it's 10100. This completes the proof, as we have found $n_0 = 1$ and $c = 10100$ such that $100n + 10000 \leq cn^2$ for all $n \geq n_0$. ■

Let's continue with an example that has negative terms. To remove a negative term, simply ensure that it really **is** negative through a suitable choice of n_0, and then remove the term. This is correct because removing a negative term from x provides an upper-bound for x.

Example. Prove that $5n^2 - 3n + 20 = O(n^2)$.

Proof. We have to show for all $n \geq n_0$ and some constant $c > 0$ that $5n^2 - 3n + 20 \leq cn^2$

For $n \geq 1$, we have

$$\begin{aligned} 5n^2 - 3n + 20 &\leq 5n^2 + 20 \\ &\leq 5n^2 + 20n^2 \\ &= 25n^2 \end{aligned}$$

So, we choose $n_0 = 1$ and $c = 25$. ■

Note that we're not forced to find the **smallest** suitable values of n_0 and c. As long as the math is correct, any suitable values of n_0 and c will do. For example, in the previous proof, we could have used $n_0 = 34$ and $c = 50$, and the proof would still be valid. (Confusing, for sure, but valid!)

Big Omega

We now know what O does: it gives an upper-bound for an expression. But remember that the O upper-bound isn't required to be tight. So if someone says that an algorithm is $O(n^3)$, you really don't know its runtime. It could be n^3, yes, but it could also be faster, like n^2 or n. So upper-bounding isn't sufficient; we'll also want to lower-bound. To lower-bound, we use a different but related notation, Ω (big Omega).

We say that $f(n)$ is **Big-Omega** of $g(n)$, written as $f(n) \in \Omega(g(n))$, iff there are positive constants c and n_0 such that

$$c\,g(n) \leq f(n) \text{ for all } n \geq n_0.$$

So, instead of upper-bounding f by g as for a O proof, we upper-bound g by f. Here's a quick example.

Example. Prove that $n^3 + 4n^2 = \Omega(n^2)$.

Proof. We have to show for all $n \geq n_0$ and some positive c that $cn^2 \leq n^3 + 4n^2$

For $n \geq 0$, we have

$$\begin{aligned} n^2 &\leq 4n^2 \\ &\leq n^3 + 4n^2 \end{aligned}$$

So, we choose $n_0 = 0$ and $c = 1$. ■

It's a common belief that O is used for worst-case analysis and Ω is used for best-case analysis. This is **not** true. Both notations can bound any function — best-case, worst-case, average-case, whatever.

Big Theta

One more notation, and then we'll be done. So far, we have a way to bound a function from above (O) and a way to bound a function from below (*Omega*). Suppose we show that $f(n) = O(n^2)$ and also that $f(n) = \Omega(n^2)$. This means that the upper-bound and lower-bound are both n^2. We then want a way to say "$f(n)$ is exactly proportional to n^2". The way to state this exact bound is using big Theta, Θ. It's a combination of O and Ω:

We say that $f(n)$ is **Big-Theta** of $g(n)$, written as $f(n) \in \Theta(g(n))$, iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Therefore, to do a Θ proof, we do two separate proofs: the O proof and the Ω proof. If the O and Ω bounds are the same, then the Θ proof is complete.

We'll start with a polynomial Θ example and then move on to a nonpolynomial example to help you generalize what you've learned about the three notations.

Example. Find a Θ bound on $f(n) = n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17$.

Proof. The highest power is n^8, so it's reasonable to try proving an n^8 Θ bound.

First, the big O proof. For $n \geq 1$:

$$\begin{aligned} n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 &\leq n^8 + 7n^7 + 3n^2 \\ &\leq n^8 + 7n^8 + 3n^8 \\ &= 11n^8 \end{aligned}$$

So, we choose $n_0 = 1$ and $c = 11$ to show that $f(n) = O(n^8)$.

Second, the big Omega proof. For $n \geq 1$:

$$\begin{aligned} n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 &\geq n^8 - 10n^5 - 2n^4 - 17 \\ &\geq n^8 - 10n^7 - 2n^7 - 17n^7 \\ &= n^8 - 29n^7 \end{aligned}$$

To continue, we'll want to show that $n^8 - 29n^7 \geq cn^8$ for some constant $c > 0$. As long as $n \geq 1$, dividing through by n^8 gives us something equivalent that we can prove:

$$\begin{aligned} n^8 - 29n^7 &\geq cn^8 \\ 1 - 29/n &\geq c \qquad \text{(if } n \geq 1\text{)} \end{aligned}$$

How can we choose c so that this inequality is true? The key is to understand the behaviour of the $-29/n$ term. $-29/n$ is increasing (substitute large values of n to observe its behaviour), which is a good thing as we want to lower-bound $1 - 29/n$. If we substitute 58 for n, $1 - 29/n = 1 - 29/58 = 0.5$, so $1 - 29/n \geq 0.5$ for $n \geq 58$. We therefore choose $n_0 = 58$ and $c = 0.5$ to complete the Ω proof. (Don't get too hung up on that choice of $n_0 = 58$: other values are possible, too. For example, we could choose $n_0 = 40, c = 0.275$.)

As the O and Ω proofs are now complete, the overall Θ proof is complete.

You might wonder why we used $n^8 - 10n^7 - 2n^7 - 17n^7$ in the proof. Why not just use $n^8 - 10n^8 - 2n^8 - 17n^8 = -28n^8$? The reason is that then we'd be stuck proving that $-28n^8 \geq cn^8$ or, equivalently for $n \geq 1$, $-28 \geq c$. We can satisfy this with something like $c = -28$ or $c = -29$, but remember that we must find a **positive** value for c, and that is impossible here. So using that 7 exponent is what allows a positive value of c to be found! ■

Example. Prove that $n^2 \in O(1.1^n)$, but that $n^2 \notin \Theta(1.1^n)$.

What are we doing here? We're being asked to prove that $c * 1.1^n$ is an upper-bound of n^2 (that's the O part), but that $d * 1.1^n$ is **not** also a lower-bound for n^2. If we can do this, then we'll have proved that $n^2 \in O(1.1^n)$, but also that $n^2 \notin \Omega(1.1^n)$. Putting those together, it follows that $n^2 \notin \Theta(1.1^n)$. This kind of proof shows that some $g(n)$ is an upper-bound, but **not** a tight upper-bound, for some function $f(n)$. There's actually a notation for this case, $f(n) = o(g(n))$ (that's a little o), but we won't use it in this course.

Proof. The O part of the proof is more annoying than previous ones, because one function is a polynomial while the other is an exponential. Informally, we could argue that an exponential grows more quickly than a polynomial, but that isn't a formal proof! Instead, you can use mathematical induction to prove that for all $n \geq 100$, $n^2 \leq 1.1^n$. Then,

using $n_0 = 100$ and $c = 1$, the O proof will be complete. This induction proof is a good exercise — please try it!

Now for the Ω part of the proof. Suppose for contradiction that, for all $n \geq n_0$ and some $c > 0$, we have
$c * 1.1^n \leq n^2$
taking logs and doing some algebra, the following lines are equivalent:

$$\begin{aligned} \ln(c * 1.1^n) &\leq \ln(n^2) \\ \ln(c * 1.1^n) &\leq 2\ln(n) \\ \ln(c) + \ln(1.1^n) &\leq 2\ln(n) \\ \ln(c) + n\ln(1.1) &\leq 2\ln(n) \\ \ln(c) &\leq 2\ln(n) - n\ln(1.1) \end{aligned}$$

but this is a contradiction: no c satisfies this equation. The reason is because the right side decreases to negative infinity. No matter what value we choose for c, the right side can always be made to be less than $\ln(c)$ through a suitably-large choice of n. ■

Properties of the Notations

The O, Ω, and Θ notations each have many properties that can be proved. These properties often follow naturally from their intuitive definitions. For example, suppose that $f(n) = O(g(n))$ and $g(n) = O(h(n))$. This says that $g(n)$ is an upper-bound for $f(n)$, and that $h(n)$ is an upper-bound for $g(n)$. You'd then expect $h(n)$ to be an upper-bound for $f(n)$ and, indeed, this can be proven using the definition of O. (In this way, O is similar to the $\leq$ operator.)

Here, we'll prove a different result: that Θ is symmetric (and, in this way, acts like the $=$ operator).

Example. Prove that if $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

Proof. The assumption is that $f(n) = \Theta(g(n))$. This tells us two things: that $f(n) = O(g(n))$ and that $f(n) = \Omega(g(n))$. Let's associate $c_1, n_1 > 0$ with the O assumption and $c_2, n_2 > 0$ with the Ω assumption. Let n_0 be the larger of n_1 and n_2.

Then, we have as assumptions that $c_2 g(n) \leq f(n) \leq c_1 g(n)$ for all $n \geq n_0$ Therefore, we have $g(n) \leq (1/c_2)f(n)$ and $g(n) \geq (1/c_1)f(n)$, for all $n \geq n_0$ Equivalently:
$(1/c_1)f(n) \leq g(n) \leq (1/c_2)f(n)$, for all $n \geq n_0$ Using constants $1/c_1$, $1/c_2$, and n_0, we see that $g(n) = \Theta(f(n))$, as required. ■