CSCE 411 Course Notes

Calvin Beideman

Fall 2026

1 Foundational Mathematical Notation

In this chapter we establish some important definitions and notation that will be used throughout the course. Hopefully many of these definitions will be familiar to you from previous courses, but there may be concepts that are new to you or slight differences between the notation used in this course and what you are familiar with.

1.1 Sets

A **set** is an unordered collection of distinct things. Often these things will be numbers, but they could also be strings, or algorithms, or other sets, or basically anything else you can think of. We call the things in a set its **elements**. Sets can be finite or infinite. We can represent a finite set with a list of its elements, separated by commas and enclosed in curly braces. So $\{1, 2, 3, 4, 5\}$ is a set, as is $\{\text{“apple”}, \text{“banana”}, \text{“orange”}\}$, as is $\{\text{“apple”}, \sqrt{3}, -100, \{\{2\}\}\}$.

The **size** or **cardinality** of a set is the number of elements that it contains. We use $|A|$ to denote the size of the set A. For example, $|\{1, 2, 5\}| = 3$ and $|\{\{\}, \{1, 2, 3, 4\}, \{7\}, \{\{5\}, \{6\}, \{9\}\}\}| = 4$. A set cannot contain multiple copies of the same element, $\{2, 2\}$ is the same set as $\{2\}$. A set can, however, have zero elements. The set $\{\}$, which contains no elements, is known as the **empty set**, and we denote it with the special symbol $\emptyset$.

We use the symbol $\in$ to say that an object is part of a set. For example $2 \in \{1, 2, 5\}$. To say that an object is not an element of a set, we can use the symbol $\notin$. For example, $4 \notin \{1, 2, 5\}$. For two sets A and B, if *all* of the elements of A are also in B, we say that A is a **subset** of B, written $A \subseteq B$ (conversely we say that B is a superset of A).

Theoretical computer science (including algorithm design and analysis) is built on discrete mathematics. Unlike ”continuous mathematics” (such as calculus) which focuses on real numbers, discrete mathematics is built on the integers. We use $\mathbb{Z}$ to denote the set of integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$. In many contexts, we don't care about negative values or they don't make sense, so we restrict ourselves to the set of non-negative integers:

$$\mathbb{Z}_{\geq 0} := \{0, 1, 2, 3, \ldots\}.$$

Note that the notation $:=$ means “is defined as”. Sometimes we don't care about 0 either, and use the set of positive integers

$$\mathbb{Z}^+ := \{1, 2, 3, \ldots\}.$$

Many authors (including CLRS) use the term “natural numbers” for $\mathbb{Z}_{\geq 0}$, and denote it $\mathbb{N}$. However, this notation can be confusing, because other authors use ”natural numbers” and $\mathbb{N}$ to refer to $\mathbb{Z}^+$. To avoid confusion over whether 0 is included in the natural numbers, these notes and your homeworks/exams will avoid the term and instead use “non-negative integers” or “positive integers” and the symbols $\mathbb{Z}_{\geq 0}$ or $\mathbb{Z}^+$.

Listing all of the elements works great for describing small sets, but if a set has hundreds of elements, or worse, infinitely many, this notation is not very effective. In the previous paragraphs we defined an infinite set using ellipses i.e. we represented the infinite set of positive integers with $\{1, 2, 3, \ldots\}$. This approach can work when the pattern that the elements follow is very obvious, but

for more complicated sets it is likely to cause confusion. A better approach is to use "set builder notation" where we explicitly describe the pattern that the elements of the set follow. This notation typically takes the form $\{x \in S \mid P(x)\}$, where P is a function outputting true or false (a predicate) and S is some set which has already been defined. This expression is read as "The set of x in S such that $P(x)$". Sometimes you will also see this notation with a colon instead of a vertical bar (i.e. $\{x \in S : P(x)\}$). Set builder notation allows us to express any subset of a known set by describing some property that all of the elements of the set satisfy. For example, we can formally describe the set of even integers as $\{x \in \mathbb{Z} \,|\, x \bmod 2 = 0\}$.

Sometimes instead of describing a subset of an existing set, we want to create a new set by applying some function f to certain elements of an existing set. For this, we can use an alternative form of set builder notation: $\{f(x) \mid x \in S \text{ and } P(x)\}$. This defines a set, which contains $f(x)$ for each $x \in S$ which satisfies the property given by P. Note that some elements of our new set may be $f(x)$ for several different $x \in S$, but a set cannot contain the same element more than once. This notation gives us an alternative way of describing the set of even integers $\{2x \mid x \in \mathbb{Z}\}$, i.e. a number is an even integer if it can be written as $2x$ where x is an integer. Similarly, we can define the set of rational numbers by:

$$\mathbb{Q} := \{p/q \mid p, q \in \mathbb{Z}, q \neq 0\}.$$

That is, a number is a rational number if it can be written as p/q where p is an integer and q is a non-zero integer.

The real numbers "fill in the gaps" of the rational numbers by including infinitely many irrational values like $\sqrt{2}$ and π. We use $\mathbb{R}$ to denote the set of real numbers. We will not formally define the real numbers or make frequent use of them in this course. Just as we denote the positive integers with $\mathbb{Z}^+$, we denote the positive rationals with $\mathbb{Q}^+$, and the positive reals with $\mathbb{R}^+$. Similarly, we denote the non-negative rationals with $\mathbb{Q}_{\geq 0}$ and the non-negative reals with $\mathbb{R}_{\geq 0}$.

1.2 Combining Sets

We saw in the previous section how to define a new set in terms of an existing set using set builder notation. We can also use this notation to define a new set in terms of two or more existing sets. Certain ways of combining sets are common enough that they have their own special notation. For two sets S and T:

- The **union** of S and T is
$$S \cup T := \{x \mid x \in S \text{ or } x \in T\}.$$
- The **intersection** of S and T is
$$S \cap T := \{x \mid x \in S \textit{ and } x \in T\}.$$
- The **difference** of S and T is
$$S \setminus T := \{x \mid x \in S \text{ and } x \notin T\}.$$
Sometimes the difference is also written as $S - T$.
- The **(Cartesian) product** of S and T is
$$S \times T := \{(x, y) \mid x \in S \text{ and } y \in T\}.$$
That is, the product of S and T is the set of all pairs where the first element is from S and the second is from T.

One final useful set operation is the **powerset** of a set S:

$$2^S := \{T \mid T \subseteq S\}.$$

For example, if $S = \{1, 2, 3\}$,

$$2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Note that if S is a finite set, $|2^S| = 2^{|S|}$. The notation $\mathcal{P}(S)$ is also sometimes used for the powerset.

1.3 Exponents and Logarithms

Exponents and logarithms arise frequently when analyzing algorithms, so you should be comfortable working with them.

For a real number x and a positive integer a, x^a is defined as the result of multiplying x with itself, a times. For example $5^3 = 5 \cdot 5 \cdot 5 = 125$. In the expression x^a, we call x the "base" and a the "exponent". We generally read x^a as "x to the a" with a few exceptions. In particular, x^2 is read as "x squared" and x^3 is read as "x cubed". Exponents satisfy the following properties which you should be familiar with. Don't treat these equations as facts to memorize—instead you should make sure that you understand *why* they're true so that when they're relevant you can apply them automatically.

Theorem 1. *If x and y are positive real numbers and a and b are positive integers then*

(a) $x^a x^b = x^{a+b}$

(b) $(x^a)^b = x^{ab}$

(c) $x^a y^a = (xy)^a$

Proof. (a) By definition x^a is the product of a copies of x and x^b is the product of b copies of x. So when we multiply them together we obtain the product of $a + b$ copies of x.

(b) $(x^a)^b$ is the product of b copies of x^a. Each x^a is the product of a copies of x. Thus, in total we are multiplying ab copies of x. Note that this equation also implies that $(x^a)^b = (x^b)^a$. Swapping the exponents like this is sometimes helpful for simplifying expressions.

(c) $x^a y^a$ is the result of multiplying x with itself a times, multiplying y with itself a times, and then multiplying the two results. Since multiplication is commutative (xy is the same as yx), we can rearrange the order of the multiplications without changing the result. In particular, we can pair up the x's and y's to rewrite our product of a copies of x and a copies of y as the product of a xy pairs, i.e. $(xy)^a$.

□

The definition of exponentiation as "repeated multiplication" makes sense for positive integer exponents, but it doesn't really make sense for other values. What would something like $5^{-2/3}$ mean? How can we multiply a number with itself negative two thirds of a time? To extend the definition of exponentiation beyond repeated multiplication, mathematicians looked at what definitions would preserve the useful properties given in Theorem 1. In particular, these properties give us values for all rational exponents as follows:

- For every real number x, we define $x^0 = 1$, because to preserve property (a) we need $x^1 \cdot x^0 = x^1$. (Note that in particular we define $0^0 = 1$, unlike in some branches of mathematics where it is considered indeterminate).
- For every non-negative real number x and every positive integer b, we define $x^{1/b}$ as the (non-negative real) solution to $y^b = x$ (i.e. the number which when multiplied with itself b times gives x). This is because to preserve property (b) we need $(x^{1/b})^b = x^1 = x$. $x^{1/b}$ is also known as "the b-th root of x" and can be written with the alternative notation $\sqrt[b]{x}$. Just as x^2 and x^3 are called "x squared" and "x cubed", we refer to $x^{1/2}$ as "the square root of x" (typically written as $\sqrt{x}$, note that we omit the 2) and we refer to $x^{1/3}$ as "the cube root of x".
- For every non-negative real number x and all positive integers a and b, we define $x^{a/b}$ as $(x^{1/b})^a$. Once again, this preserves property (b).
- For every positive real number x and every positive rational number a, we define $x^{-a} = 1/x^a$. This is because to preserve property (a) we need $x^a x^{-a} = x^0 = 1$.

We've now defined x^a whenever x is a positive real number and a is a rational number. We could extend our definition to cover every real number a using limits, but the details are outside the focus of this course (since we will not need tools from calculus elsewhere). While we will omit the formal definition, you may assume in this course that the properties in Theorem 1 hold for all real numbers a and b.

Logarithms are the inverse of exponents. $\log_b x$ is defined as the power that you have to raise b to in order to get x, i.e. $\log_b x$ is the value of y for which $b^y = x$.

The following theorem gives some properties of logarithms which you should understand and be comfortable using. Part (c) in particular allows us to convert a logarithm from one base to another, which can be useful for analyzing divide and conquer algorithms that we will discuss later in the course.

Theorem 2. *If a is a real number and b, c, x, and y are positive real numbers with $b, c \neq 1$ then*

(a) $\log_b(xy) = \log_b(x) + \log_b(y)$

(b) $\log_b(x^a) = a \log_b(x)$

(c) $\log_b x = (\log_c x) \log_b c$

Proof. (a)

$$\begin{aligned} b^{\log_b(x)+\log_b(y)} &= b^{\log_b(x)} b^{\log_b(y)} && \text{by part (a) of Theorem 1} \\ &= xy && \text{by the definition of log} \end{aligned}$$

Thus, since $b^{\log_b(x)+\log_b(y)} = xy$, by the definition of log, $\log_b(xy) = \log_b(x) + \log_b(y)$.

(b)

$$\begin{aligned} b^{a \log_b(x)} &= (b^{\log_b(x)})^a && \text{by part (b) of Theorem 1} \\ &= x^a && \text{by the definition of log} \end{aligned}$$

Thus, since $b^{a \log_b(x)} = x^a$, by the definition of log, $\log_b(x^a) = a \log_b(x)$.

(c)

$$\begin{aligned} b^{(\log_c x) \log_b c} &= (b^{\log_b(c)})^{\log_c x} && \text{by part (b) of Theorem 1} \\ &= c^{\log_c x} && \text{by the definition of log} \\ &= x && \text{by the definition of log} \end{aligned}$$

Thus, since $b^{(\log_c x) \log_b c} = x$, by the definition of log, $\log_b x = (\log_c x) \log_b c$.

□

When we have an exponent where the base is a logarithm, we will often use the following shorthand notation to avoid extra parentheses

$$\log_b^k n := (\log_b n)^k.$$

For example, $(\log_2 5)^3$ can be written as $\log_2^3 5$.

1.4 Summations and Products

We use the symbol $\sum$ (the capital Greek letter sigma) to represent a summation. You can think of this like a mathematician's version of a for loop. If f is some function which has been defined on integers and a and b are integers, then the expression

$$\sum_{i=a}^{b} f(i)$$

is equivalent to running the pseudocode

```
sum ← 0
For i from a to b:
   sum ← sum +f(i)
Return sum
```

For example, $\sum_{i=3}^{5} i^2 = 3^2 + 4^2 + 5^2 = 50$ (note that the upper and lower bounds for our iteration variable i can be written either above and below the $\sum$, as in the first example, or just to the right, as in the example in this paragraph).

Sometimes we want to iterate over something other than a range of integers. $\sum$ notation can also be used for looping over the elements of an arbitrary set:

$$\sum_{x \in S} f(x)$$

This expression represents applying the function f to every element of the set S and adding up the results.

Just as we use $\sum$ to denote sums, we use $\prod$ (the capital Greek letter pi) to denote products. The use is exactly the same, except with $\prod$ notation we are *multiplying* the results of each iteration of the loop instead of adding them. For example, $\prod_{i=3}^{5} i = 3 \cdot 4 \cdot 5 = 60$.

The summation in the following theorem shows up frequently enough in analyzing algorithms that you should be familiar with it and know how to simplify it.

Theorem 3.

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

Proof. We note that

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

Since addition is commutative (we can rearrange the order in which we add numbers without changing the sum), we can rewrite this sum as

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

This new version has n pairs of values and each pair adds up to $n+1$. Thus,

$$2\sum_{i=1}^{n} i = n(n+1),$$

and

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

□

We'll learn how to simplify some other summations in Chapter 3.

1.5 Other Important Mathematical Notation

For a real number x, we use $\lfloor x \rfloor$ to denote the *floor* of x, which is the largest integer less than or equal to x. Similarly we use $\lceil x \rceil$ to denote the *ceiling* of x, which is the smallest integer greater than or equal to x. For example:

- $\lfloor 4.5 \rfloor = 4$
- $\lceil 4.5 \rceil = 5$
- $\lfloor -2.7 \rfloor = -3$
- $\lceil -2.7 \rceil = -2$
- $\lfloor 7 \rfloor = 7$
- $\lceil 7 \rceil = 7$

For a non-negative integer n, we use $n!$ to denote n *factorial*, which is $n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$. For example, $5! = 120$. We define $0! := 1$.

1.6 Arrays and Strings

In this class, we will use many of the abstract data types and data structures you learned about in CSCE 221. The most important abstract data type is the **list** ADT, which represents an *ordered* collection of objects. Unlike a set, a list may contain multiple copies of the same value.

Lists are commonly implemented using the **array** data structure. The core property of arrays in contrast to other implementations of the list ADT (like linked lists) is that arrays allow for random access, meaning that we can get the element at any arbitrary position in the array in constant time, without having to first look through earlier elements of the array. As in many programming languages, we will use square brackets to denote indexing into an array. For example, $A[5]$ refers to the element at index 5 in the array A. Unlike some programming languages you may be familiar with (such as C++), in this course we will typically view arrays as beginning at index 1. Therefore we will use $A[1, \ldots, n]$ to denote an arbitrary array of size n.

We will sometimes use $A[i : j]$ to refer to the subarray starting at index i and ending at index j, i.e. $A[i, i+1, \ldots, j-1, j]$. For example, if A is the array $[3, 1, 4, 1, 5, 9]$ then $A[1] = 3$, and $A[2 : 4] = [1, 4, 1]$. Note that this convention is different from some programming languages like Python where $A[i : j]$ does not include $A[j]$.

An **alphabet** is a finite, nonempty set. We call the elements of the alphabet **symbols** or **characters**. Common choices in this course will be the English alphabet (the set containing the letters a through z) and the alphabet $\{0, 1\}$. We often use the character Σ as a variable for an alphabet (note that this is the same Greek letter used for describing summations, do not confuse the two uses). A **string** x over the alphabet Σ is a finite sequence of symbols $x_1 x_2 x_3 \ldots x_n$, where $x_i \in \Sigma$ for each $1 \leq i \leq n$. We typically write the symbols in the string right next to each other, but we can add commas or spaces if this would be ambiguous. We use $|x|$ to denote the length or size of x, which is the number of symbols that it contains. We use Σ^n to denote the set of all length-n strings over the alphabet Σ. We use Σ^* to denote the infinite set of all strings over the alphabet Σ (of any length). Thus $0011010 \in \{0, 1\}^7$, and therefore also $0011010 \in \{0, 1\}^*$ (we call strings over the alphabet $\{0, 1\}$ **binary strings**), and if Σ is the English alphabet then

$$\text{hello} \in \Sigma^*.$$

The empty string is a valid string over any alphabet, and we denote it with the special character ε. Given two strings x and y over alphabets Σ and Γ, their **concatenation** is the string over the alphabet $\Sigma \cup \Gamma$ which consists of all the symbols of x followed by all the symbols of y. We denote x concatenated with y as $x \cdot y$ or simply xy. For example, if x is the binary string 010 and y is the binary string 1011, xy is the binary string 0101011. We use x^n to denote the string obtained by

concatenating x with itself n times. For example, $(01)^4$ is 01010101. Be careful not to confuse this with exponentiation. To determine whether x^n means "x to the power n" or "the concatenation of n copies of x" you need to know whether x is being treated as a number or a string.

1.7 Comprehension Questions

1. What is the cardinality of the set $\{\{3, 7, 11\}, \{A, B\}\}$?
2. What is $|\{x \in \mathbb{Z} \mid x^2 < 10\}|$? (Note: don't confuse this with the set $\{x \in \mathbb{Z}_{\geq 0} \mid x^2 < 10\}$!)
3. What is $\{1, 3, 5\} \setminus \{1, 2, 3\}$?
4. What is $\{1, 2\} \times \{3, 4\}$?
5. What is $2^{\{1,2\}}$?
6. Let x and n be positive integers greater than 1 such that $(x^2x^3)^5 = x^n$. What is the value of n?
7. What is the value of $\log_2(8^{32})$?
8. What is the value of $\sum_{i=1}^{100} i$?
9. What is the value of $\prod_{i=1}^{5} i$?
10. What is the value of $\lfloor -2.1 \rfloor \cdot \lceil 3 \rceil$?
11. Which of these strings are in $\{0, 1\}^3$?
    - ε
    - 01
    - 101
    - 012
    - 1000
12. Let $\Sigma = \{a, b, c, d\}$, and consider the two strings ab and cd. What is $ab \cdot cd$?
13. Let $\Sigma = \{0, 1\}$, and consider the two strings 10 and 100. What is $10 \cdot 100$?
14. Compute $|\{s \in \{0, 1, 4\}^3 : s_1 = s_2\}|$.

1.8 Exercises

1. Let n be a positive integer greater than 1 and let x be a positive real number such that $\left(\frac{3}{2}\right)^{\log_2 n} n = n^x$. What is the value of x?
2. Let a, b, c, x, y be integers greater than 2 and let α and β be real numbers such that

   $$\log_b(x^a y^c) = \alpha \log_c(x) + \beta \log_b(y).$$

   What are the values of α and β? (Express your answer in terms of numeric constants and/or a, b, and c).
3. Let n be an integer greater than 99. Simplify

   $$\sum_{i=99}^{2n} (i - 3)$$

   (i.e. give an expression which is equivalent to this one but doesn't use a summation).

2 Logic and Proofs

This chapter will review mathematical logic and basic techniques for formally proving mathematical claims. These proof techniques will be essential for proving the correctness and runtime of the algorithms we will develop and analyze in this course. Hopefully much of this content is familiar to you from CSCE 222, but it's important to establish a common foundation to ensure that everyone is set up for success when we begin applying these proof techniques to new topics.

2.1 Predicate Logic

A **predicate** is some statement which is true or false. For example $10 < 25$, $19^2 = 400$, and "The string 'strawberry' contains 2 r's" are all predicates (the first one is true, the other two are false). On the other hand an expression like $2 + 2$ is not a predicate, since it does not express a claim and therefore does not have a truth value. Similarly, $x > -2$ is not a predicate, because x is not defined, meaning that the statement does not have a well-defined truth value.

If we want to include variables in our predicate, we need to **quantify** them, meaning we need to describe what values the variable can take on. There are two main logical quantifiers.

- The universal quantifier "for all" is used to say that some expression is true for every element of a given set. We could convert our expression $x > -2$ into the (false) predicate "For all integers x, $x > -2$" by adding a universal quantifier. Note that the choice of which set to quantify over is very important. We could convert the same expression into the true predicate "For all positive integers x, $x > -2$" by choosing to quantify over $\mathbb{Z}^+$ instead of $\mathbb{Z}$.

  We represent the universal quantifier symbolically using the symbol $\forall$. So the predicate we just described could be written as $\forall\, x \in \mathbb{Z}^+, x > -2$.

  Note that any "for all" statement is true if the set that we are quantifying over is the empty set. For example

  $$\forall\, x \in \emptyset,\ 2 + 2 = 5$$

  is a true predicate! We say that a predicate universally quantifying over the empty set is **vacuously true**.

- The existential quantifier "there exists" is used to say that some expression is true for at least one element of a given set. We could convert our expression $x > -2$ into the (true) predicate "There exists an integer x such that $x > -2$" by adding an existential quantifier. Once again, the choice of set to quantify over is important. We could make a false predicate based on the same expression if we chose to quantify over the set $\{-3, -4, -5\}$.

  We represent the existential quantifier symbolically using the symbol $\exists$. For example: $\exists\, x \in \mathbb{Z}, x > -2$.

  Just as every "for all" statement about the empty set is true, every "there exists" statement about the empty set is false. For example

  $$\exists\, x \in \emptyset,\ 2 + 2 = 4$$

  is a false predicate. There is no x in the empty set for which $2 + 2 = 4$, because there is no x in the empty set at all!

We can modify and combine predicates with the logical operators "and" (denoted symbolically as $\wedge$), "or" (denoted symbolically as $\vee$), and "not" (denoted symbolically as $\neg$). So if A and B are two predicates then:

- $A \wedge B$ is true if A and B are both true and false otherwise.
- $A \vee B$ is true if either A or B is true, and false if A and B are both false. Note that in mathematics/computer science "or" almost always means "inclusive or", so $A \vee B$ is true if A and B are both true.

- $\neg A$ is true if A is false, and false if A is true.

The logical operators in the previous paragraph all correspond to common operators in many programming languages. There is another important logical operator which is not common in programming: implication. "A implies B" is a predicate which is true if A and B are both true, or if A is false. This predicate can also be written as "If A, then B" or denoted symbolically as $A \Rightarrow B$. Many of the predicates we are interested in proving are implications.

Implications can be confusing, because the way they are used in mathematics and computer science is different from the way that "if" is used in regular English. It's important to remember that $A \Rightarrow B$ is always true when A is false. For example "If I am on Mars, then $2 + 2 = 5$" is a true predicate. Since I am not on Mars, the truth value of $2 + 2 = 5$ does not affect the truth value of the whole expression (If you are reading this from a Martian colony in the distant future, replace Mars with a planet you are not on).

Closely related to implication is **bi-implication**, also known as "if and only if". "A bi-implies B" or "A if and only if B" or "A iff B" all mean that A and B are either both true or both false. Bi-implication is written symbolically as $A \Leftrightarrow B$. $A \Leftrightarrow B$ is logically equivalent to $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

2.2 What is a Proof?

A **proof** is a rigorous argument designed to convince other people that a given claim is true. A correct proof could be written fully in predicate logic, relying only on basic axioms of set theory. Such proofs are very tedious to write and to read, however, so typical proofs are written mostly in English. This means that what constitutes a "correct" proof is to some degree a social convention. A correct proof will never claim a false statement, but a proof which makes only true statements can still be considered incorrect if it skips too many steps and it's not clear how the lines of the proof connect to each other.

For example, let's say that I want to prove the following uninteresting theorem:

Theorem 4. $((19 \cdot 15) + 7)^2 - 9 > 10000$

Proof. We first calculate that

$$19 \cdot 15 = 285.$$

Therefore

$$(19 \cdot 15) + 7 = 285 + 7 = 292.$$

Thus,

$$((19 \cdot 15) + 7)^2 = 292^2 = 292 \cdot 292 = 85264.$$

Finally,

$$((19 \cdot 15) + 7)^2 - 9 = 85264 - 9 = 85255 > 10000.$$

□

Alternatively, we could give the following shorter proof of the same theorem:

Proof. We note that

$$((19 \cdot 15) + 7)^2 - 9 = 85255 > 10000.$$

□

For the purposes of this course, the second proof would be entirely sufficient. You may assume that the reader of your proof is capable of performing basic arithmetic and do not need to show every step of the calculation. On the other hand, you should not assume that your reader is comfortable manipulating complicated summations, and if you need to do this, you should carefully show the steps.

How much detail your proof should go into depends on the audience that you are writing for. A proof written for CS PhD's may skip many steps as "obvious" that should be fully justified if the proof is targeted at CS undergraduates. You should view the audience of the proofs you write in

this class as a fellow student in this course who is not quite as clever as you. You may assume basic results from prerequisite courses like Calculus and CSCE 222 without proving them, and you may use any theorem proven in these notes. The original parts of your proof, however, should be carefully justified. You want to make an airtight case which would convince a reader who is skeptical of the claim you are proving.

One good test for deciding whether you need to justify something in your proof is to think about what you would say if someone asked you why a particular line of the proof follows from the previous one. If you can think of a short obvious answer, the connection is probably clear enough. If you realize that *you* couldn't explain the connection, then that's a sign that your proof is missing something there.

When in doubt, err on the side of providing more detail. We will not penalize you for providing an overly rigorous proof, but we will penalize you for leaving out important details. You are also welcome to ask in office hours if you have justified a particular part of your proof sufficiently.

2.3 Proving Predicates

In the previous section we saw an example of how to prove the simplest kind of predicate, a formula with no variables or quantifiers. Most predicates we are interested in proving, however, will involve a universal or existential quantifier (or both).

To prove a universal statement of the form $\forall\, x \in S,\ P(x)$, we need to choose an *arbitrary* $x \in S$ and show that $P(x)$ holds. Here "arbitrary" means that our proof should work for any choice of x from S, we can't assume that the x we chose has any special property. Sometimes it's helpful to imagine here that x is being chosen for you by an enemy who wants your proof to fail.

For example, suppose that we want to prove the following theorem:

Theorem 5. *Let* $S := \{n^2 \mid n \in \mathbb{Z}^+\}$. *Then* $\forall\, x \in S,\ x > 1 \Rightarrow x^2 > x + 10$

Proof. Let $x \in S$ be arbitrary. If $x \leq 1$ the implication is automatically true, so assume $x > 1$. Since $x \in S$, x is the square of a positive integer. Since the next smallest square number after 1 is 4, $x > 1 \Rightarrow x \geq 4$. Therefore:

$$x^2 \geq 4x = x + 3x \geq x + 12 > x + 10.$$

□

We can view this kind of proof as providing a template for creating infinitely many proofs, one for each element of S. We can pick any element of S to be x, plug it into this proof, and the proof will show us why $x > 1 \Rightarrow x^2 > x + 10$ for that value of x. Note that in this proof we explicitly considered the case where our arbitrary x was less than or equal to 1. While it is important to understand that we are proving a claim about all x, including values of x which are less than or equal to 1, this step is typically omitted in proofs of "for all" statements including implications, because it would be the same every time (every implication is true whenever its left side is false). Therefore we can shorten the proof slightly by starting "Let $x \in S$ be an arbitrary integer with $x > 1$".

Proving a "there exists" statement is usually easier. To prove $\exists\, x \in S,\ P(x)$, we simply need to find a single specific $x \in S$ for which $P(x)$ is true. For example:

Theorem 6. *Let* $S := \{n^2 \mid n \in \mathbb{Z}^+\}$. *Then* $\exists\, x \in S,\ x^2 < \log_2(x) + 10$

Proof. Choose $x = 1$. $x \in S$, since $1 \in \mathbb{Z}^+$ and $1^2 = 1$. For this choice of x, we have:

$$x^2 = 1^2 = 1 < 10 = 10 + 0 = 10 + \log_2(1).$$

□

Sometimes we will want to prove statements involving *nested quantifiers*, i.e. statements that have both "for all" and "there exists" statements. When parsing these statements it is important to pay attention to the order of the quantifiers. To prove a statement of the form $\forall\, x \in S,\ \exists\, y \in T,\ P(x, y)$, we need to show that for any arbitrary x in S, we can pick some y in T such that $P(x, y)$ is true (the choice of y will probably depend on the value of x).

For example,

Theorem 7. $\forall x \in \mathbb{Z}, \exists y \in \mathbb{Z}^+, y > x$.

Proof. Let $x \in \mathbb{Z}$ be arbitrary. If $x < 0$, choose $y = 1$, and then $y \in \mathbb{Z}^+$ and $y > x$. Otherwise, $x \geq 0$, so choose $y = x + 1$. Since $x \geq 0$, $y \in \mathbb{Z}^+$, and $y = x + 1 > x$. □

Notice that we had to pay attention to the sets that x and y were quantified over here. It would be easy to simply say "choose $y = x + 1$" without considering a special case for negative x, but since the theorem says that y must be positive, that proof would be incorrect.

If we switch the order of the quantifiers in Theorem 7, we get a different predicate

$$\exists y \in \mathbb{Z}^+, \forall x \in \mathbb{Z}, y > x.$$

This predicate claims that there exists a single y, such that for every integer x, y is larger than that x. Since the existential quantifier comes first now, we have to choose y before x, meaning that our choice of y cannot depend on x like it did in the proof of Theorem 7. There is no integer which is larger than all other integers, so this predicate is false.

2.4 Disproving Predicates

Sometimes we want to disprove a predicate (i.e. to show that it is false). This is equivalent to proving the negation of the predicate. Therefore, it's important to understand how to negate universal and existential quantifiers.

The negation of the statement $\forall x \in S, P(x)$ is $\exists x \in S, \neg P(x)$. This means that to disprove a "for all" statement, it's enough to find a single counterexample. Finding one $x \in S$ for which $P(x)$ is false, proves that $P(x)$ can't be true for every single $x \in S$.

The negation of the statement $\exists x \in S, P(x)$ is $\forall x \in S, \neg P(x)$. That is, to show that a "there exists" statement is false, we need to prove that $P(x)$ is false for every single x in S. We can prove this using any of our usual techniques for proving a "for all" statement, like the "choose an arbitrary element and show the claim holds for it" method that we saw in the previous section or induction which we will talk about in Section 3.1.

2.5 Using Cases

Often in a proof we need to divide a claim up into several cases and prove each of them separately. For example, to prove a statement of the form $A \vee B \Rightarrow C$, we may have one case where we assume A is true, and then prove C, and a second case where we assume B is true, and then prove C. Note that these cases may overlap (it's possible that A and B are both true), which is not a problem. The important thing is that our cases are comprehensive. Any situation where $A \vee B$ is true will fall into at least one of our cases.

Cases can also be helpful when proving a "for all" statement. As we've seen, the standard approach to proving a statement of the form $\forall x \in S, P(x)$ is to take an arbitrary $x \in S$ and show that $P(x)$ holds. Sometimes we need a different proof structure depending on some property of the arbitrary x we chose (for example, we might use a different proof for odd x than for even x). This is another natural place to use case work. Again it's important to justify that every $x \in S$ will fall into at least one of our cases. Here is an example of using casework to prove a "for all" statement.

Theorem 8. *Let $S := \{n^2 \mid n \in \mathbb{Z}_{\geq 0}\}$. Then $\forall x \in S$, the last digit of x is either 0, 1, 4, 5, 6, or 9.*

Proof. Let $x \in S$ be arbitrary. By the definition of S, x is a perfect square, so $\sqrt{x} \in \mathbb{Z}_{\geq 0}$. Let $n := \sqrt{x}$. We note that the last digit of n^2 depends only on the last digit of n (this follows from the grade school algorithm for multiplication). We case on the last digit of n.

- If the last digit of n is 0, then the last digit of n^2 is 0
- If the last digit of n is 1, then the last digit of n^2 is 1
- If the last digit of n is 2, then the last digit of n^2 is 4

- If the last digit of n is 3, then the last digit of n^2 is 9
- If the last digit of n is 4, then the last digit of n^2 is 6
- If the last digit of n is 5, then the last digit of n^2 is 5
- If the last digit of n is 6, then the last digit of n^2 is 6
- If the last digit of n is 7, then the last digit of n^2 is 9
- If the last digit of n is 8, then the last digit of n^2 is 4
- If the last digit of n is 9, then the last digit of n^2 is 1

These are all the possible values for the last digit of n. Thus, in every case, the last digit of n^2 (which by definition equals x) is either 0, 1, 4, 5, 6, or 9. □

2.6 Using the Contrapositive

Remember that an implication is a statement of the form "If A then B", written $A \Rightarrow B$. The standard way to prove an implication is to start by assuming A is true and then use this to show that B is true (perhaps by applying definitions or performing algebraic manipulation).

Sometimes it's easier to prove an implication if we first convert it into a different equivalent form. One of the most common ways to do this is by taking the **contrapositive**. The contrapositive of $A \Rightarrow B$ is $\neg B \Rightarrow \neg A$. Just like the original implication the contrapositive is false only when A is true and B is false, so the two forms are logically equivalent, and we can choose to prove whichever is more convenient. Here is an example of a proof by contrapositive:

Theorem 9. $\forall\, x, y \in \mathbb{Z}_{\geq 0},\ x + y > 21 \Rightarrow x > 10 \vee y > 11.$

Proof. Let x and y be arbitrary non-negative integers. We want to prove that $x + y > 21 \Rightarrow x > 10 \vee y > 11$. We will prove the contrapositive:

$$\neg(x > 10 \vee y > 11) \Rightarrow \neg(x + y > 21).$$

By De Morgan's Laws, this is equivalent to

$$x \leq 10 \wedge y \leq 11 \Rightarrow x + y \leq 21.$$

If $x \leq 10$ and $y \leq 11$, then $x + y \leq 10 + 11 = 21$, which proves the theorem. □

Here it was not quite clear how to directly prove the implication starting from the fact $x + y > 21$ and trying to prove an "or" statement, but the proof of the contrapositive is very short and simple. Note that taking the contrapositive only affected the implication, not the quantifiers, we were still proving a "for all" statement about non-negative integers.

2.7 Proving Subset Relationships and Set Equality

To prove that one set S is a subset of another set T we need to prove the predicate

$$\forall\, x \in S,\ x \in T.$$

Just like proving any other "for all" statement, this will typically involve taking an arbitrary element of S, and then showing that it is in T (usually by expanding the definitions of S and T and showing how they relate to each other).

Proving that two sets are equal requires a little more effort. The most common approach is a proof by **double containment**, which means showing $S \subseteq T$ and $T \subseteq S$ (if both of these are true, then we must have $S = T$). Here is an example of a set equality proof (which includes two subset proofs):

Theorem 10. *Let $S := \{25x + 15y \mid x, y \in \mathbb{Z}\}$, and let $T := \{5n \mid n \in \mathbb{Z}\}$. Prove that $S = T$.*

Proof. We prove this by double containment:

- First we show $S \subseteq T$. Let $a \in S$ be arbitrary. By the definition of S, a can be expressed as $25x + 15y$ for some integers x and y. We need to show that there exists an integer n such that $a = 5n$. Let $n := 5x + 3y$. Then we have that

  $$5n = 5(5x + 3y) = 25x + 15y = a.$$

  So by the definition of T, $a \in T$.

- Now we show $T \subseteq S$. Let $a \in T$ be arbitrary. By the definition of T, $a = 5n$ for some integer n. We need to show that there exist integers x and y such that $25x + 15y = a$. We note that $25(-1) + 15(2) = 5$. Let $x := -n$ and let $y := 2n$ (clearly these are both integers). Then we have that

  $$25x + 15y = 25(-n) + 15(2n) = -25n + 30n = 5n = a.$$

  Thus, by the definition of S, $a \in S$.

□

If you are asked to prove set equality it is important to remember to prove both directions.

2.8 Acknowledgments

Some of the topics and problems in this chapter are inspired by ones in Building Blocks for Theoretical Computer Science by Margaret Fleck, the textbook that I used to teach Discrete Structures at UIUC.

2.9 Comprehension Questions

1. Determine which of the following predicates are true and which are false:

   (a) $3 > 4 \vee 3 > 2$

   (b) $3 > 4 \wedge 3 > 2$

   (c) $3 > 4 \Rightarrow 3 > 2$

   (d) $3 > 2 \Rightarrow 3 > 4$

   (e) $\forall x \in \mathbb{Z}, x \neq 4.11$

   (f) $\exists x \in \mathbb{Z}, x \neq 4$

   (g) $\forall x \in \{-3, 3\}, x^2 = 9$

   (h) $\exists x \in \mathbb{Z}^+, x \leq 0$

   (i) $\forall x \in \emptyset, x^2 = -1$

   (j) $\exists x \in \emptyset, x = x$

   (k) $\forall x \in \mathbb{Z}, x > 4 \Rightarrow x^2 > x$

   (l) $\forall x \in \mathbb{Z}, x > 0 \Rightarrow 2 + 2 = 4$

   (m) $\forall x \in \mathbb{Z}^+, 2 + 2 = 4 \Rightarrow x > 0$

   (n) $\exists x \in \mathbb{Z}^+, 2 + 2 = 4 \Rightarrow x = 7$

   (o) $\exists x \in \mathbb{Z}^+, x > 7 \Rightarrow x = -3$

   (p) $\exists x \in \mathbb{Z}^+, x > 0 \Rightarrow x = 5$

   (q) $\exists x \in \mathbb{Z}^+, x > 0 \Rightarrow x = -3$

   (r) $\forall x \in \mathbb{Z}^+, x > 7 \Rightarrow x \neq 1$

2. Find the negation of each of the following predicates

   (a) $\forall\, x \in \mathbb{Z},\, \exists\, y \in \mathbb{Z}^+,\, x + y = 7$

   (b) $\exists\, x \in \mathbb{Z},\, \forall\, y \in \mathbb{Z}^+,\, x + y = 7$

   (c) $\exists\, x, y \in \mathbb{Z}^+, x + y = 7$

   (d) $\forall\, x \in \mathbb{Z}_{\geq 0}, x > 5 \Rightarrow x \neq 3$

3. Find the contrapositive of each of the following implications:

   (a) $x > 5 \Rightarrow x^2 > 10$

   (b) x^2 is odd $\Rightarrow x$ is odd

   (c) $x \notin A \cap B \Rightarrow x \notin A$

4. True or False? For all sets A and B, $A = B \Leftrightarrow (A \subseteq B \wedge B \subseteq A)$.

5. Fill in the blank: When we convert the implication $A \Rightarrow B$ to the equivalent implication $\neg B \Rightarrow \neg A$, we have taken the _____.

6. Suppose that we want to prove "$\forall\, x \in \mathbb{R}^+, x^2 > 0$." What is the main flaw with the following "proof"?

   Proof. Let $x \in \mathbb{R}^+$ be arbitrary. Then we have that

   $$x^2 = x \cdot x \geq x \geq 0.$$

   □

7. Suppose that we want to prove that $\{2n \mid n \in \mathbb{Z}_{\geq 0}\} = \{n \in \mathbb{Z}_{\geq 0} \mid$ the last digit of n is either $0, 2, 4, 6$ or $8\}$." What is the main flaw with the following "proof"?

   Proof. Let $S := \{2n \mid n \in \mathbb{Z}_{\geq 0}\}$ and let $T := \{n \in \mathbb{Z}_{\geq 0} \mid$ the last digit of n is either $0, 2, 4, 6$ or $8\}$. Let $x \in S$ be arbitrary. Then $x = 2n$ for some $n \in \mathbb{Z}_{\geq 0}$. We case on the last digit of n:

   - If the last digit of n is 0 or 5, the last digit of $x = 2n$ is 0, so $x \in T$.
   - If the last digit of n is 1 or 6, the last digit of $x = 2n$ is 2, so $x \in T$.
   - If the last digit of n is 2 or 7, the last digit of $x = 2n$ is 4, so $x \in T$.
   - If the last digit of n is 3 or 8, the last digit of $x = 2n$ is 6, so $x \in T$.
   - If the last digit of n is 4 or 9, the last digit of $x = 2n$ is 8, so $x \in T$.

   Since n must end with one of these digits, we conclude that in every case $x \in T$ and thus $S = T$. □

2.10 Exercises

1. Prove that $\forall\, a \in \mathbb{Z}, \exists\, b, c \in \mathbb{Z}, a \neq b \wedge a \neq c \wedge b + c = a$.

2. Prove that $\forall\, a, b \in \mathbb{Z},\ a < b \Rightarrow \exists\, q \in \mathbb{Q},\ a < q < b$.

3. Prove that $\forall\, x \in \mathbb{Q}^+, \forall\, y \in \mathbb{R}, \frac{y+1}{2} \in \mathbb{Q} \Rightarrow \frac{1}{x} + y \in \mathbb{Q}$

4. Prove that if $r \in \mathbb{R}^+ \setminus \mathbb{Q}^+$ and $q \in \mathbb{Q}^+$, then $rq \in \mathbb{R} \setminus \mathbb{Q}$ (i.e. prove that multiplying a positive irrational number with a positive rational number gives an irrational number).

5. Define a new operation $\ominus$ by $a \ominus b := 2(a + b)$. **Disprove** the following claim:
   Claim: $\forall\, x, y, z \in \mathbb{R}, x \ominus (y \ominus z) = (x \ominus y) \ominus z$

6. Let A, B, and C be sets. Prove that

$$(A \setminus B) \cup (B \setminus C) \subseteq (A \cup B) \setminus (A \cap B \cap C)$$

7. Prove that for every positive integer n, if n^2 is even, then n is even.

8. Let Σ be an alphabet. Prove that

$$\{x \cdot y \mid x, y \in \Sigma^* \text{ with } |x| = |y|\} = \{s \in \Sigma^* \mid |s| \text{ is even }\}$$

3 Advanced Proof Techniques

This chapter will discuss some more advanced proof techniques, building on what we discussed in the previous section. We'll start in Section 3.1 by reviewing induction, a powerful technique you've probably seen before. In Section 3.2, we introduce correctness proofs for algorithms, the main kind of proof we'll be using in this course, and in Section 3.3, we'll introduce loop invariants, a way to prove the correctness of certain programs using induction. Finally in Section 3.4, we'll discuss proof by contradiction, a powerful technique for proving that particular objects (or algorithms) can't exist, which we'll rely on later in the course, particularly in Chapter 7.

3.1 Induction

One of the most powerful proof techniques is induction. Induction is a technique for proving a "for all" statement quantified over non-negative integers. It is based on the important axiom that every non-empty subset of $\mathbb{Z}_{\geq 0}$ has a minimum element.

An induction proof proves a statement of the form $\forall\, n \in \mathbb{Z}_{\geq 0}$, $P(n)$ by proving

$$\forall\, n \in \mathbb{Z}_{\geq 0}\, (\forall\, k \in \mathbb{Z}_{\geq 0} \text{ with } k < n,\, P(k)) \Rightarrow P(n)$$

Here we have replaced the original predicate $P(n)$ with the new predicate "If $P(k)$ for all $0 \leq k < n$, then $P(n)$". Since we have given ourselves an additional assumption to work with, this new predicate is often easier to prove. This form of induction is often known as "strong" induction, in contrast to "weak" induction where we only assume $P(n-1)$ instead of assuming $P(k)$ for all $0 \leq k < n$. There is no particular benefit to using weak induction, however, so I encourage you to always use the form presented above.

It may not be clear however why proving the inductive version of the statement implies the original statement. To see why induction is valid, it is helpful to think about it as giving us a template to create a proof of $P(n)$ for whatever value of n we want. For $n = 0$, the assumption "$P(k)$ holds for all $0 \leq k < n$" is vacuously true (there are no possible choices of k which are ≥ 0 but also < 0), so our assumption has given us nothing and our induction proof must directly prove $P(0)$ (we often call this a base case). For $n = 1$, our induction proof shows that $P(0) \Rightarrow P(1)$. Since we already have a direct proof that $P(0)$ is true, we can combine these to get a proof that $P(1)$ is true (we know that $P(0)$ is true, and we know "if $P(0)$ is true then $P(1)$ is true", so we conclude $P(1)$ is true). Similarly, our induction proof shows that $P(0) \wedge P(1) \Rightarrow P(2)$. Since we have now shown $P(0)$ and $P(1)$, we can combine these proofs together with the proof that $P(0) \wedge P(1) \Rightarrow P(2)$ to get a proof that $P(2)$ is true. Continuing the pattern, we can use our induction template to construct a direct proof of $P(n)$ for any given n by first constructing direct proofs for $P(0), \ldots, P(n-1)$.

Don't despair even if the correctness of induction feels suspect, however. It's possible to construct correct induction proofs even if the fact that induction works still feels like magic.

Let's see an example of the kind of claim that we can prove with induction.

Theorem 11. *$\forall\, n \in \mathbb{Z}_{\geq 0}$, $n^3 - n$ is divisible by 3*

Proof. We prove this by induction on n.

- Base Case: If $n = 0$, then $n^3 - n = 0^3 - 0 = 0$, which is divisible by 3.
- Induction Hypothesis: Suppose that $\forall\, k \in \mathbb{Z}_{\geq 0}$ with $k < n$, $k^3 - k$ is divisible by 3.
- Induction Step: We will show that $n^3 - n$ is divisible by 3. Since $n > 0$, $n - 1 \in \mathbb{Z}_{\geq 0}$, so by our induction hypothesis we have that $(n-1)^3 - (n-1)$ is divisible by 3. We note that

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

  Therefore $n^3 - n - 3(n^2 - n)$ is divisible by 3. Since n is an integer, $3(n^2 - n)$ is divisible by 3 as well, so we conclude that $n^3 - n$ is divisible by 3.

□

This proof illustrates the main pieces of an induction proof.

- First we explicitly state what variable we are inducting on. Sometimes this will be obvious, but other times the choice of induction variable will be trickier (we may need to induct on the sum or difference of two values for example).
- Next we have the base case, where we show the claim for 0 (or whatever the smallest value the claim applies to is) directly
- Then we state our induction hypothesis, which is just assuming that the claim holds for all values of k less than n (We don't literally have to call our variables n and k, but it's important that we call the variable we're assuming the claim holds for something different from the variable we are inducting on. If we're inducting on n and assume that the claim holds for n, our induction hypothesis is already assuming what we're trying to prove in the induction step!)
  - Sometimes it may be convenient to instead assume the claim holds for values up to and including n, then prove it for $n+1$. This is also fine, as long as you use the same form in the induction hypothesis and induction step).
- Finally, we have the induction step, which is usually the meat of the proof. This is where we prove that our claim holds for n using the induction hypothesis. (If your induction step does not use your induction hypothesis either you did something wrong or you didn't need to be using induction in the first place)

At its core, induction is a technique for proving universal statements about non-negative integers. We can apply induction to many other kinds of problems however, by converting them into universal statements about non-negative integers. The key is usually choosing an appropriate non-negative integer variable to induct on. Here is an example of using induction to prove a theorem about strings:

Theorem 12. *Let Σ be an alphabet and let $x, y \in \Sigma^*$ be non-empty strings such that $xy = yx$. Then there exists a string $z \in \Sigma^*$ and integers $a, b \in \mathbb{Z}^+$ such that $x = z^a$ and $y = z^b$.*

Proof. We prove this by induction on $|xy|$.

- Base case: $|xy| = 2$ (this is the smallest possible length for xy since x and y are nonempty). In this case, x and y must both have length 1, so since $xy = yx$, we have that $x = y$. Therefore we can choose $z = x$ and $a = b = 1$, and we have that $z^a = x$ and $z^b = y$.
- Induction Hypothesis: Assume that for all $u, v \in \Sigma^*$ with $|uv| < |xy|$, $uv = vu \Rightarrow \exists\, z \in \Sigma^*, \exists\, a, b \in \mathbb{Z}^+,\ u = z^a \wedge v = z^b$.
- Induction Step: We consider two cases:

  Case 1: Suppose $|x| = |y|$. Since $xy = yx$, we know that x is the same as the first $|x|$ many characters of y, and since $|y| = |x|$, this is all of y, so $x = y$. Therefore we can choose $z = x$ and $a, b = 1$.

  Case 2: Otherwise, $|x| \neq |y|$. Assume without loss of generality that $|x| > |y|$. Since $xy = yx$, the first $|y|$ characters of x are a copy of y, so we can write x as yw, for some non-empty string $w \in \Sigma^*$. Substituting this into our equality $xy = yx$, we get that $ywy = yyw$. Peeling off y from the front of both strings, this implies that $wy = yw$. Furthermore, since $|y| > 0$, $|wy| < |xy|$. Therefore, by our induction hypothesis, there exists a string $z \in \Sigma^*$ and integers $a, b \in \mathbb{Z}^+$ such that $w = z^a$ and $y = z^b$. For the same z, we have that $x = yw = z^b z^a = z^{a+b}$, so x and y can both be created by concatenating copies of z, which is what we wanted to show.

□

Note that this induction proof would actually be complete without the base case. The case where $|xy| = 2$ is already covered in the inductive step by our special case for when $|x| = |y|$. There's no real harm in covering this case twice, though, so it's fine to include a base case even when it's technically not needed.

This proof also used the useful phrase "without loss of generality". This phrase (sometimes abbreviated WLOG) is essentially a shortcut for doing a proof by cases where the proofs for the different cases are exactly the same except with the names of the variables swapped. In this case, rather than giving separate proofs for the case where $|x| > |y|$ and the case where $|y| > |x|$, we arbitrarily choose to assume $|x| > |y|$, and use the phrase "without loss of generality" to let the reader know that they could obtain the proof for the other case by simply swapping x and y in the rest of the proof.

Back in Section 1.4 we saw a simple proof that $\sum_{i=1}^{n} i = n(n+1)/2$. This is probably the most useful summation formula to know for analyzing algorithms, but the second most useful is the one we'll prove now, using induction.

Theorem 13. *For all $n \in \mathbb{Z}_{\geq 0}$ and all $x \in \mathbb{R} \setminus \{1\}$,*

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Proof. We prove this by induction on n.

- Base Case: If $n = 0$, then since $x \neq 1$ (and thus $x - 1 \neq 0$) we have

$$\sum_{i=0}^{0} x^i = x^0 = 1 = \frac{x-1}{x-1} = \frac{x^{0+1} - 1}{x - 1}$$

- Induction Hypothesis: Assume that for all $k \in \mathbb{Z}_{\geq 0}$ with $k < n$, $\sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1}$
- Induction Step: We have that

$$\begin{aligned}
\sum_{i=0}^{n} x^i &= x^n + \sum_{i=0}^{n-1} x^i && \text{(Since } n > 0\text{)} \\
&= x^n + \frac{x^n - 1}{x - 1} && \text{(by our IH)} \\
&= \frac{x^{n+1} - x^n}{x - 1} + \frac{x^n - 1}{x - 1} \\
&= \frac{x^{n+1} - 1}{x - 1}
\end{aligned}$$

□

Note that we split the summation from 0 to n into the last term plus a summation from 0 to $n - 1$ so that we could apply the induction hypothesis. This is a standard technique that we can use to apply induction to many summations. Finding the right formula for a summation can be tricky, but if we have a guess at the right answer (e.g. based on finding a pattern in the value of the summation for small values of n) this technique will usually let us prove that the guess is correct.

3.2 Proving the Correctness of Algorithms

The most common proofs you will need to write in this course are proofs of correctness for algorithms. A proof of correctness for an algorithm can be complicated and may need to be broken down into several pieces, but at the highest level it's just a proof of a "for all" statement, similar to the ones we've seen in sections 2.3, 2.5 and 3.1. In particular, proving an algorithm means proving that for all

inputs the algorithm produces the correct output. For example, if SORT is an algorithm designed to sort an array of integers (in order of increasing value), then to prove the correctness of SORT we need to prove that:

"For every array A of integers, after calling SORT(A), A is sorted."

Sometimes correctness proofs are simple enough that we don't need any special techniques. Consider the following algorithm which finds the square root of any number whose square root is an integer:

```
IntegerSquareRoot(n):
    i ← 0
    While (i + 1)^2 ≤ n:
        i ← i + 1
    Return i
```

Theorem 14. *For every non-negative integer n such that $\sqrt{n}$ is an integer,* INTEGERSQUAREROOT(n) *returns $\sqrt{n}$.*

Proof. Let $n \in \mathbb{Z}_{\geq 0}$ such that $n = k^2$ for some $k \in \mathbb{Z}_{\geq 0}$. For every $i < k$, $(i+1)^2 \leq k^2 = n$, while if $i = k$ then $(i+1)^2 > k^2$. Thus, the while loop will continue until $i = k$. Thus, when the algorithm returns i it is returning $k = \sqrt{n}$. □

Note that if we want to prove an algorithm correct it is important to be clear about what inputs the algorithm is expected to handle. INTEGERSQUAREROOT would not be an algorithm that correctly returns $\sqrt{n}$ if the input could be any integer (e.g. the algorithm fails to compute $\sqrt{2}$, because $\sqrt{2}$ is not an integer). While data validation is important in real-world programming, the correct way to handle invalid inputs tends to be application specific. In this course, if you are asked to design an algorithm that takes a specific kind of input (e.g. a square number), your algorithm may do whatever it likes when the input is not of the type given in the problem, and you do not need to consider these kinds of inputs when proving the correctness of your algorithm.

For recursive algorithms, induction is often a natural approach to proving correctness (we'll look at this more when we talk about divide and conquer algorithms in Chapter 8). For example, consider the following algorithm:

```
Fact(n):
    If n ≤ 1:
        Return 1
    Return n * Fact(n − 1)
```

Theorem 15. *For every positive integer n,* FACT(n) *returns $n!$ (recall from Section 1.5 that $n!$ is n factorial)*

Proof. We prove this by induction on n:

- Base Case: If $n = 1$ then FACT immediately returns 1, which is 1!
- Induction Hypothesis: Suppose that for every $1 \leq k < n$, FACT(k) returns $k!$.
- Induction Step: Since $n > 1$, FACT(n) returns $n \cdot$ FACT($n-1$). By our induction hypothesis, FACT($n-1$) returns $(n-1)!$. Thus, FACT(n) returns $n(n-1)! = n!$.

□

3.3 Loop Invariants

The algorithms we saw in the previous section were so simple that they arguably don't even need a full proof of correctness (While it's always nice to rigorously prove things, in an exam where functions like INTEGERSQUAREROOT or FACT might be a subroutine in a larger program it would be sufficient to say that their correctness follows from the definition of square root or factorial). Now let's look at an algorithm whose correctness proof is a bit less trivial.

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Theorem 16. *For every array A of integers, after calling* InsertionSort*(A), A is sorted in increasing order.*

Before proving Theorem 16, let's think a bit about what the right approach is. We need to prove that InsertionSort(A) sorts A. Induction is the right idea here, but we have to be careful about what we induct on and what claim we prove inductively. Unlike in our correctness proof for Fact we can't induct on the input directly because the input to InsertionSort is an array, not an integer. We could try inducting on the length of A, but since InsertionSort doesn't recursively call itself, it's not clear how to use an induction hypothesis about the behavior of InsertionSort on arrays of length $< n$ to say something about the behavior of InsertionSort on arrays of length n.

Instead, the correct approach is to induct on the number of iterations of the outer for loop. Our induction hypothesis can't be that the array is sorted (clearly it need not be sorted after just one iteration of the loop), but that each iteration makes increasing progress towards sorting the array, in such a way that by the end of the n'th iteration, the array is sorted. A claim of the form: "After i iterations of this loop, $P(i)$ is true" (where $P(i)$ is some predicate depending on i) is called a *loop invariant*. Defining a suitable loop invariant and then proving it (usually via induction) is a common technique for proving the correctness of algorithms.

Let's prove the correctness of insertion sort using a loop invariant:

Proof. (of Theorem 16) Let A be an array of integers.

We will prove the following loop invariant for every $k \in \{1, \ldots, n\}$: After k iterations of the outer for loop, $A[1:k]$ is sorted. Note that this invariant implies that after n iterations of the for loop (i.e. at the end of the algorithm), $A[1:n]$ is sorted, meaning the entire array has been sorted. Thus, proving this invariant is sufficient to prove the theorem.

We will prove the invariant by induction on k.

- Base Case: Suppose $k = 1$. The first iteration of the outer loop does nothing (because the inner loop runs 0 times). However, $A[1, \ldots, k] = A[1, \ldots, 1]$ consists of just a single element, so it is automatically sorted.

- Induction Hypothesis: Suppose that for every $1 \leq \ell < k$, we have that $A[1, \ldots, \ell]$ is sorted after ℓ iterations of the outer for loop.

- Induction Step: By our induction hypothesis, we know that at the start of the k'th iteration (i.e. after the $(k-1)$'th iteration) $A[1:k-1]$ is sorted. Let x be the value in $A[k]$ at the beginning of the k'th loop iteration. In the k'th iteration of the outer loop, the inner loop repeatedly swaps x with elements of the sorted subarray $A[1, \ldots, k-1]$, starting from the right, until either it reaches the beginning of the array, or it finds an element which is less than or equal to x.

  If we swap x all the way to the beginning of the array, then we know that x is smaller than every other element of $A[1:k]$ and thus belongs in $A[1]$, where it has ended up. Meanwhile all of the elements $A[1:k-1]$ have shifted right one position, but they remain in the same relative order. By our induction hypothesis they were already sorted, so therefore $A[1:k]$ is now sorted.

  On the other hand if we stop swapping x and break out of the loop at some $j > 1$, then we found some j such that $A[j] \leq x$, and our new array begins $A[1:j], x, A[j+1, k-1]$. Since we broke out of the inner for loop for this value of j, but not the previous one, we have that $A[j] \leq x < A[j+1]$. Furthermore, by our IH, $A[1:j]$ and $A[j+1:k-1]$ were already sorted,

so in the rearranged array we have $A[m] \leq A[m+1]$ for all m from 1 to $k-1$, and $A[1:k]$ is sorted.

□

3.4 Proof by Contradiction

The last type of proof it's important to be familiar with is proof by contradiction. In a proof by contradiction we start by assuming the opposite of what we're trying to prove and then show that this implies some statement which we know is false. Just like with induction, it's important to let the reader know when you're using this proof technique, so a proof by contradiction usually begins with "Assume for the sake of contradiction that [opposite of claim you're trying to prove]". "Assume for sake of contradiction" can be abbreviated as AFSOC.

A proof by contradiction is complete when you have shown that the thing you have assumed implies a false statement. Often this will be a contradiction to what you assumed at the start of the proof (e.g. you assumed a number was odd then proved it was even), but it can also be a regular logical impossibility, like deriving that $1 = 2$.

Proof by contradiction is commonly used when we want to show that a particular object (for example, an algorithm) cannot exist. One of the earliest examples of this kind of proof comes from the Greek mathematician Euclid, over 2000 years ago, who proved that there are infinitely many primes (Euclid's proof was not explicitly a proof by contradiction, but the core idea of it is the same as the proof I present here).

Theorem 17. *There is no largest prime number*

Proof. AFSOC that there is a largest prime number, call it p. Let n be the number of prime numbers less than or equal to p, and call these primes $p_1, \ldots, p_n$, with $p_n = p$. Let $m := \prod_{i=1}^{n} p_i$. By definition, m is divisible by every prime less than or equal to p. Therefore, $m+1$ is not divisible by any of these primes, meaning that either $m+1$ is prime, or it is divisible by some other prime bigger than p. In either case, this contradicts our assumption that p is the largest prime (because $m+1 > m \geq p$). Therefore, there is no largest prime number. □

Here's another classic example of proof by contradiction. Note that the claim "$\sqrt{2}$ is irrational" is actually a claim of the form "an object does not exist" in disguise. If we unpack the definition of a rational number, saying that "$\sqrt{2}$ is not a rational number" means saying that there do not exist two integers whose ratio is $\sqrt{2}$. Therefore we can prove this claim by contradiction if we assume that there are two such integers, and then show that this implies nonsense.

Theorem 18. $\sqrt{2}$ *is irrational.*

Proof. AFSOC that $\sqrt{2} \in \mathbb{Q}$. Then $\exists\, p, q \in \mathbb{Z}$ such that $\sqrt{2} = p/q$ and p and q are relatively prime (i.e. the fraction p/q is in simplest terms). We have that $2 = (p/q)^2 = p^2/q^2$, so $2q^2 = p^2$. Therefore p^2 is even, which means that p must also be even (since squaring an odd number gives an odd number). Thus, $p = 2r$ for some integer r. Since we already concluded that $2q^2 = p^2$, this means we now have $2q^2 = 4r^2$, and thus $q^2 = 2r^2$, so q^2 is also even, and therefore q is even as well. But this contradicts our assumption that p and q were relatively prime! (they share a factor of 2, so p/q can be simplified). We have reached a contradiction, so our initial assumption that $\sqrt{2} \in \mathbb{Q}$ must be false, and $\sqrt{2}$ is irrational. □

Now that we've discussed a bunch of valid proof techniques, I want to leave you with a word of warning about an invalid "proof" technique that students sometimes use. While showing that $\neg P \Rightarrow$ False is a valid way to prove that P is true, showing that $P \Rightarrow$ True does *not* mean that P is true (remember that any implication with a false premise is true). For example, if I start from $1 = 2$, I can multiply both sides by 0 to obtain the true statement $0 = 0$, but clearly this does not imply that $1 = 2$.

3.5 Acknowledgments

Some of the topics and problems in this chapter are inspired by ones in Building Blocks for Theoretical Computer Science by Margaret Fleck, the textbook that I used to teach Discrete Structures at UIUC.

3.6 Comprehension Questions

1. Suppose that we want to prove the claim "For every positive integer n, $n^2 + n$ is even." by induction on n. Which of the following would be an appropriate induction hypothesis to use?

   (a) Assume that for every positive integer k, $k^2 + k$ is even.

   (b) Assume that for every integer $k < n$, $k^2 + k$ is even.

   (c) Assume that for every positive integer $a < n$, $a^2 + a$ is even.

   (d) Assume that $n^2 + n$ is even.

   (e) Assume that for every integer n with $0 \leq n < k$, $n^2 + n$ is even.

2. Given that $2^{20} = 1048576$, what is the value of $\sum_{i=1}^{19} 2^i$?

3. Suppose that we design an algorithm ALG to solve the following problem:

   Problem: Given a positive integer n, determine whether or not n is a power of 2 (i.e. determine whether or not $\log_2 n$ is an integer).

   Which of the following statements would we need to prove in order to show that ALG correctly solves the problem? (select all that apply)

   (a) For every positive integer n which is a power of 2, ALG(n) returns true.

   (b) For every positive integer n which is not a power of 2, ALG(n) returns false.

   (c) For every non-positive integer n, ALG(n) returns false.

   (d) For every non-positive integer n, ALG(n) reports that the input is invalid.

4. Which of the following loop invariants would be most appropriate to use in proving the correctness of the sorting algorithm below?

```
ShiftySort(A[1, ..., n]):
  For i from 1 to n − 1:
      For j from n − i to n − 1:
          If A[j] > A[j + 1]:
              Swap A[j] with A[j + 1]
          Else:
              Break
```

   (a) After k iterations of the outer for loop, $A[1, \ldots, k]$ is sorted.

   (b) After k iterations of the outer for loop, $A[1, \ldots, k + 1]$ is sorted.

   (c) After k iterations of the outer for loop, $A[n - k + 1, \ldots, n]$ is sorted.

   (d) After k iterations of the outer for loop, $A[n - k, \ldots, n]$ is sorted.

   (e) After k iterations of the outer for loop, $A[n - k - 1, \ldots, n]$ is sorted.

5. Find the main flaw in the following proof (it's not enough to give a counterexample to the claim, you should explain why the induction proof fails).

   Claim: All cats are the same color.

   Proof: We will show that for any positive integer n, for every set of n cats, all those cats are the same color. Since the size of the set of all cats on earth is some positive integer n,

this suffices to prove the claim. The proof is by induction on n.

Base Case: $n = 1$. Certainly, in a group of one cat, all cats have the same color (the color of that single cat). In order to have two cats of different colors we need at least two cats.

Induction Hypothesis: For every $k < n$, for every group of k cats, all cats in the group are the same color.

Induction Step: Let S be an arbitrary set of n cats. Let $c \in S$ be an arbitrary cat in the set. We want to show that every other cat in S has the same color as c. Let d be another cat in S. By the induction hypothesis, all of the cats in the set $S - \{c\}$ have the same color. Also by the induction hypothesis, all of the cats in the set $S - \{d\}$ have the same color. Therefore, both c and d are the same color as all of the cats in $S - \{c, d\}$, and therefore c and d are the same color as each other. Since c and d were chosen arbitrarily, this proves that all the cats in S are the same color.

3.7 Exercises

1. Use induction to show that

$$\forall\, n \in \mathbb{Z}^+ \quad \sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

2. Use induction to show that

$$\forall\, n \in \mathbb{Z}^+, \quad \sum_{i=1}^{n} \lfloor \log_7 i \rfloor = \lfloor \log_7 n \rfloor (n+1) - \frac{7^{\lfloor \log_7 n \rfloor + 1} - 7}{6}$$

3. Use induction to show that

$$\forall\, n \in \mathbb{Z}_{\geq 0}, \quad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

4. Prove that $\sqrt{3}$ is irrational.

5. Prove that $\log_2 3$ is irrational.

6. Consider the following algorithm:

```
RPM(x, y)
    r ← 0
    a ← x
    b ← y
    While b > 0:
        If b mod 2 = 1 :
            r ← r + a
        a ← 2a
        b ← ⌊b/2⌋
    Return r
```

Prove that for every $x, y \in \mathbb{Z}_{\geq 0}$, $\mathrm{RPM}(x, y) = xy$, i.e. prove that RPM multiplies its inputs.

7. We define a function $f(n) : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ as follows:

$$\begin{aligned} f(0) &= 2 \\ f(1) &= 7 \\ f(k) &= f(k-1) + 2f(k-2) \qquad\qquad \forall\, k \geq 2 \end{aligned}$$

   Prove that $\forall\, n \in \mathbb{Z}_{\geq 0},\ f(n) = 3 \cdot 2^n + (-1)^{n+1}$

8. Prove that the sorting algorithm below is correct (i.e. that given an array of integers, after running the algorithm the array will be sorted in ascending order). Hint: Start by showing that the algorithm always terminates.

```
ExchangeSort(A[1, ..., n])
    Repeat forever:
        i ← −1
        For j from 1 to n − 1:
            If A[j] > A[j + 1]:
                i ← j
        If i = −1:
            //A is now sorted
            Return
        Swap A[i] with A[i + 1]
```

4 Asymptotic Analysis

You've almost certainly seen Big-O notation in a previous CS course. But you may not be familiar with the formal definitions of Big-O or its relatives Big-Ω, Big-Θ, little-o, and little-ω, and you probably don't have a lot of experience using these definitions in proofs.

In Section 4.1 we formally define Big-O notation and give some examples of using it to prove one function is Big-O of another. In Section 4.2 we will introduce related notation and show how to apply it. Finally in Section 4.3 we'll discuss some useful properties of Big-O and see some examples of more abstract proofs with these definitions.

4.1 Big-O

While you're probably familiar with using Big-O to describe the runtime of algorithms, it's actually a way to compare the growth rates of arbitrary functions. Informally, for functions f and g we say that one function $f(n) \in O(g(n))$ if $f(n)$ is upper bounded by a constant multiple of $g(n)$ for all large enough n.

Formally, for a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

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

(Note that in a slight abuse of notation, we use "$\forall\, n \geq n_0$" to mean "$\forall\, n \in \mathbb{Z}$ such that $n \geq n_0$". The fact that n is an integer is implied by the fact that f is defined as a function on integers)

That is, $O(g(n))$ is the set of all functions f for which we can choose positive constants c and n_0 such that for every integer $n \geq n_0$ (i.e. every sufficiently large input), $f(n) \leq cg(n)$ ($f(n)$ is upper bounded by a constant multiple of $g(n)$). You can think of $O(g(n))$ as the set of all functions whose "dominant term" is less than or equal to $g(n)$'s (up to constant factors). For example, the set $O(2n^2 - 1)$ includes the following functions (among infinitely many others):

- $2n^2 - 1$ (we can choose $c = 1$ and $n_0 = 1$)
- $3n^2$ (we can choose $c = 3$ and $n_0 = 3$)
- $1000n$ (we can choose $c = 1000$ and $n_0 = 1$)

To prove that $f(n) \in O(g(n))$ for some specific functions f and g, we need to apply the definition. We need to show that f satisfies the condition for membership in $O(g(n))$, i.e. that $\exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \leq cg(n)$. As we saw in Section 2.3, to prove a "there exists" statement, we need to find specific values for the variables which make the statement true. In this case that means that we need to give a specific c and n_0 such that $f(n) \leq cg(n)$ for every $n \geq n_0$ (and then justify that the c and n_0 we chose actually do satisfy this predicate).

Let's see a couple examples.

Theorem 19.

$$3n^3 + 5n^2 - n + 14 \in O\left(\frac{n^3}{2}\right)$$

Proof. We need to choose positive constants c and n_0 such that $\forall\, n \geq n_0$, $3n^3 + 5n^2 - n + 14 \leq cn^3/2$. Let $c = 44$ and $n_0 = 1$. For every $n \geq 1$, we have $n^3 \geq n^2 \geq n \geq 1$, so

$$3n^3 + 5n^2 - n + 14 \leq 3n^3 + 5n^3 + 14n^3 = 22n^3 = 44 \cdot \frac{n^3}{2}.$$

□

Note that we don't need to choose the smallest possible value of c. Rather, pick whichever value of c makes the proof easiest. In the previous proof we chose $c = 44$, because it was twice the sum of the non-negative coefficients in $3n^3 + 5n^2 - n + 14$, allowing us to upper bound each term in the polynomial with a multiple of n^3. We also didn't need to explain why we chose this c in the

proof. Remember that the purpose of a proof is to convince your reader that a claim is true, not to explain how you discovered that the claim is true. Therefore a Big-O proof only needs to show that particular values of c and n_0 work, not explain how you came up with them.

Sometimes there isn't a natural inequality that we can use to directly prove the claim we want. In these cases, we may need to use induction.

Theorem 20.

$$2^n \in O(n!)$$

Proof. Let $n_0 = 1$ and $c = 2$. We will prove by induction on n that $2^n \leq 2 \cdot n!$ for every $n \geq 1$.

Base Case: As a base case, suppose $n = 1$. Then

$$2^n = 2 = 2 \cdot 1 = 2 \cdot 1! = 2 \cdot n!$$

Induction Hypothesis: Suppose that for all positive integers $k < n$, $2^k \leq 2 \cdot k!$.

Induction Step: Suppose $n \geq 2$. Applying the induction hypothesis to $n - 1$, we have that

$$\begin{aligned}
2^n &= 2 \cdot 2^{n-1} \\
&\leq 2 \cdot 2 \cdot (n-1)! && \text{(by our IH)} \\
&\leq 2 \cdot n \cdot (n-1)! && \text{(since } n \geq 2\text{)} \\
&= 2 \cdot n! && \text{(by the definition of factorial)}
\end{aligned}$$

□

In both of the previous examples, we were able to choose $n_0 = 1$ as long as we picked a large enough c. This almost always works, however if $g(n)$ sometimes outputs 0, then we usually need to pick n_0 large enough to guarantee that $g(n) > 0$ for all $n \geq n_0$. Note that $g(n)$ cannot output negative values (for positive integer inputs) because we've defined big-O only for functions with co-domain $\mathbb{R}_{\geq 0}$ (it's possible to extend the definition to functions with negative outputs, but those functions aren't very relevant in this course, since it's generally impossible for an algorithm to use a negative amount of resources).

Theorem 21.

$$\log_2(2n + 2) \in O(\log_3(n))$$

Proof. Let $n_0 = 2$ and $c = 6$. Then we have that for every $n \geq n_0$,

$$\begin{aligned}
\log_2(2n+2) &\leq \log_2(4n) && \text{(because } n \geq 1\text{)} \\
&= \log_2(4) + \log_2(n) && \text{(by Theorem 2)} \\
&= \log_2(n) + 2 \\
&\leq 3\log_2(n) && \text{(because } n \geq 2, \text{ so } \log_2(n) \geq 1\text{)} \\
&= 3\log_2(3)\log_3(n) && \text{(by Theorem 2)} \\
&\leq 3\log_2(4)\log_3(n) \\
&= 6\log_3(n)
\end{aligned}$$

□

Note that we couldn't choose $n_0 = 1$ here, because $\log_2(2 \cdot 1 + 2) = 2$, while $\log_3(1) = 0$, and 2 is not less than $0 \cdot c$ for any value of c.

Let's look at a trickier big-O proof.

Theorem 22.

$$\log_2^2 n \in O(n)$$

Proof. Let $n_0 = 1$ and $c = 49$. We will prove that $\log_2^2 n \leq 49n$ for all $n \in \mathbb{Z}^+$ by induction on n.

Base Cases: If $1 \leq n < 128$ then

$$\log_2^2 n < \log_2^2 128 = 7^2 = 49 = 49 \cdot 1 \leq 49 \cdot n.$$

Induction Hypothesis: Suppose that for all positive integers $k < n$, $\log_2^2 k \leq 49k$.

Induction Step: Let $n \geq 128$. Then

$$\begin{aligned}
\log_2^2(n) &= (\log_2(n/2) + 1)^2 && \text{(by Theorem 2)} \\
&= \log_2^2(n/2) + 2\log_2(n/2) + 1 \\
&\leq \log_2^2(n/2) + 3\log_2(n/2) && \text{(since } n \geq 2 \text{ so } \log_2 n \geq 1\text{)} \\
&= \left(1 + \frac{3}{\log_2(n/2)}\right) \log_2^2(n/2) \\
&\leq (3/2)\log_2^2(n/2) && \text{(because } n > 128 \Rightarrow \log_2(n/2) \geq 6 \Rightarrow 3/\log_2(n/2) \leq 1/2\text{)} \\
&\leq (3/2)\log_2^2(\lceil n/2 \rceil) \\
&\leq (3/2)(49)\lceil n/2 \rceil && \text{(by our IH)} \\
&\leq 49 \cdot \frac{3}{2} \cdot \frac{n+1}{2} \\
&= 49 \cdot \frac{3n+3}{4} \\
&\leq 49n && \text{(because } n \geq 3\text{)}
\end{aligned}$$

Thus, $\log_2^2 n \leq 49n$, as desired. □

There are several things to note about this proof. The choice of $c = 49$ and the base case of $n < 128$ may seem like arbitrary magic numbers, but they actually come from thinking about the structure of our induction proof.

The starting point for developing this proof was the logic of the induction step. We need to relate $\log_2^2(n)$ to $\log_2^2(k)$ for some $k < n$. $\log_2(n)$ doesn't have a clear relationship with $\log_2(n-1)$, but we do know a nice relationship between $\log_2(n)$ and $\log_2(n/2)$ (namely $\log_2(n) = \log_2(n/2) + 1$). Unfortunately, we can't necessarily apply our induction hypothesis to $n/2$, because $n/2$ may not be an integer. That's why before applying the IH we need to upper bound $n/2$ with $\lceil n/2 \rceil$, which is an integer.

If we can say that $\log_2^2(n) \leq (1+\varepsilon)\log_2^2(\lceil n/2 \rceil)$, for some $\varepsilon > 0$ then we can apply the IH to get that

$$\log_2^2(n) \leq (1+\varepsilon)c\lceil n/2 \rceil \leq (1+\varepsilon)c \cdot \frac{n+1}{2}.$$

In order to upper bound this with cn, we need $\varepsilon < 1$ and n sufficiently large. We could pick any value $0 < \varepsilon < 1$ and calculate a corresponding value of c that would make the proof work, but $\varepsilon = 1/2$ is an easy choice. In order to upper bound $3/\log_2(n/2)$ with $1/2$, we need $n \geq 128$, which is why we made $1 \leq n < 128$ our base case.

Assuming $n \geq 128$, the calculations in our induction step actually go through for any choice of c. The reason we chose $c = 49$ was because $49 = \log_2^2 128$, and this makes the proof of the base case simple.

Note that we could not have chosen $n_0 = 128$ using this same proof setup. In order to assume the induction hypothesis at $n = 128$, we need the base case to have proved our claim for all $1 \leq k < 128$. In particular, our proof uses the fact that the inequality holds for $n = 64$ to prove that it holds for $n = 128$, which would be a problem if we didn't have base case covering $n = 64$ (we could validly prove $\log_2^2(n) \in O(n)$ using $n_0 = 64$ and $c = 1$ by tweaking this proof slightly, I encourage you to think about how).

One final note on notation. Often you will see $f(n) \in O(g(n))$ written as $f(n) = O(g(n))$. This notation is slightly misleading, since, unlike equality, the big-O relationship is not commutative (e.g.

$n = O(n^2)$, but $n^2 \neq O(n)$), but it's very widespread, so you should understand it (As computer scientists, you should already be used to the idea of a one-way equals, since many programming languages use = for assignment. For example, in C++, "x = 5" is valid code whereas "5 = x" is not).

4.2 Other Asymptotic Comparisons

Big-O functions sort of like a $\leq$ relation for functions. $O(g(n))$ is the set of all functions which are eventually less than or equal to $g(n)$, up to constant factors. It's natural to also define a corresponding notion of $\geq$ for functions. This is called Big-Ω (read "Big Omega"). Similarly we have a version of "=", for functions. We use Big-Θ (read "Big Theta") to denote the set of functions which grow at the same rate as $g(n)$ up to constant factors.

Formally, for a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$\Omega(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \geq cg(n)\}$$

(this is just the big-O definition with $f(n) \leq cg(n)$ changed to $f(n) \geq cg(n)$), and

$$\Theta(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \exists\, c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

(here we need f to be bounded on both sides by constant multiples of g).

Note that the definition of big-Ω immediately implies that if $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$, and vice-versa (we can use the same values for c and n_0 in both definitions).

Theorem 23.

$$\forall\, a, b \in \mathbb{Z}^+ \text{ such that } a, b \geq 2, \log_a(n) \in \Theta(\log_b n)$$

Proof. Let a and b be positive integers greater than or equal to 2. Let $c_1 = c_2 = \log_a b$, and let $n_0 = 2$. Then, by Theorem 2, $\log_a n = c_1 \log_b n = c_2 \log_b n$, so for all $n \geq n_0$, we have

$$c_1 \log_b n \leq \log_a n \leq c_2 \log_b(n),$$

so $\log_a(n) \in \Theta(\log_b n)$ □

By Theorem 23, the base of a logarithm doesn't really matter from an asymptotic perspective (it only changes the value of the function by a constant factor), so when dealing with logarithms in Big-O and related notations, we often omit the base and simply write $O(\log n)$. Do be careful with this, however. If a logarithm shows up inside of an exponent, the base is relevant again, $O(\log_2 n)$ is the same as $O(\log_4 n)$, but $O(4^{\log_4 n})$ is the same as $O(n)$, while $O(4^{\log_2 n})$ is the same set as $O(n^2)$.

Finally, we have notions of $<$ and $>$ for functions, which we call little-o and little-ω ("little omega"). For a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$o(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \forall\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \leq cg(n)\}$$

to be the functions that are "less than" $g(n)$, asymptotically. Note that we have changed the "$\exists\, c$" from the Big-O definition into a "$\forall\, c$". This definition is saying that for any possible choice of c, no matter how tiny, $f(n)$ will eventually become smaller than $cg(n)$ (and stay smaller for all large enough n).

Similarly, for a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$\omega(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \forall\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \geq cg(n)\}$$

to be the functions which grow faster than $g(n)$ by more than a constant factor.

Let's see an example of using these new definitions:

Theorem 24.

$$\frac{n^3}{10} - 3n^2 + 1000 \in \omega(n^2)$$

Proof. We need to show that for every possible choice of c, there exists some n_0 such that for every $n \geq n_0$, $\frac{n^3}{10} - 3n^2 + 1000 \geq cn^2$.

Let c be an arbitrary positive real number. Choose $n_0 = \lceil 10(c+3) \rceil$, so that we know n_0 is an integer with $n_0 \geq 10(c+3)$. Then we have

$$n^3/10 - 3n^2 + 1000 > n^3/10 - 3n^2 = (n/10 - 3)n^2 \geq (n_0/10 - 3)n^2 \geq ((10(c+3))/10 - 3)n^2 = cn^2.$$

□

4.3 Properties of Big-O and Friends

In this section we'll show a variety of useful properties that will allow us to express the asymptotic behavior of new functions using the behavior of functions we've already seen.

Theorem 25. *Let $f, g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$. Then $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.*

Proof. To prove an "if and only if" statement, we need to show both directions.

First we prove the forward direction ($f(n) \in \Theta(g(n)) \Rightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$). Suppose $f(n) \in \Theta(g(n))$. Let $c_1, c_2 \in \mathbb{R}^+$, $n_0 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$ (these exist by the definition of big-Θ). Then, by the definition of big-O, using the constants c_2 and n_0, $f(n) \in O(g(n))$, and by the definition of big-Ω, using the constants c_1 and n_0, $f(n) \in \Omega(g(n))$.

Now we prove the back direction ($f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Theta(g(n))$). Suppose $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. Let $c_1 \in \mathbb{R}^+$, $n_1 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_1, f(n) \geq c_1 g(n)$ (these constants exist by the definition of big-Ω), and let $c_2 \in \mathbb{R}^+$, $n_2 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_2, f(n) \leq c_2 g(n)$ (these constants exist by the definition of big-O). Then, if we let $n_0 = \max(n_1, n_2)$ (so that $n \geq n_0 \Rightarrow n \geq n_1$ and $n \geq n_2$) we have that $\forall\, n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$, so $f(n) \in \Theta(g(n))$. □

Theorem 26. *If $f_1, f_2, g_1, g_2 : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ are functions such that $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then:*

(a) $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$

(b) $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

Proof. (a) Let $c_1, c_2 \in \mathbb{R}^+$, $n_1, n_2 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ and $\forall\, n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$ (such constants must exist by the definition of big-O). To show that $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ we choose the constants $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$. Then, by our choice of c_1, c_2, n_1, and n_2, we have that for every $n \geq n_0$

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq (c_1 + c_2) g_1(n) + (c_1 + c_2) g_2(n) = (c_1 + c_2)(g_1(n) + g_2(n)).$$

So by the definition of big-O, $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

(b) Let $c_1, c_2 \in \mathbb{R}^+$, $n_1, n_2 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ and $\forall\, n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$ (such constants must exist by the definition of big-O). To show that $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$ we choose the constants $c = c_1 \cdot c_2$ and $n_0 = \max(n_1, n_2)$. Then, by our choice of c_1, c_2, n_1, and n_2, we have that for every $n \geq n_0$

$$f_1(n) \cdot f_2(n) \leq c_1 g_1(n) \cdot c_2 g_2(n) = c(g_1(n) \cdot g_2(n)).$$

So by the definition of big-O, $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

□

If a problem specifically asks you to show that one function is big-O (or big-Ω, etc.) of another using the formal definition (like in the exercises for this chapter), then you should make sure to give an explicit choice of c and n_0 and justify why they work. When using Big-O to describe the runtime of your algorithms, however, you do not need to go into the full details of a big-O proof. If you've concluded that the runtime is $O(n^2) + O(n)$, you can and should say that this is $O(n^2)$ without further proof.

4.4 Comprehension Questions

1. Suppose that we want to prove that $3n^2 + 10n + 2 \in O(n^2)$ using the formal definition of big-O. Which of the following values for c and n_0 could we choose? (select only one)

   (a) $c = 10, n_0 = 1$
   (b) $c = 7, n_0 = 3$
   (c) $c = 5, n_0 = 4$
   (d) $c = 3, n_0 = 100$

2. Suppose that we want to prove that $n! \in O(n^n - 1)$ using the formal definition of big-O. Which of the following values for c and n_0 could we choose? (select only one)

   (a) $c = 0.5, n_0 = 3$
   (b) $c = 0.6, n_0 = 2$
   (c) $c = 1, n_0 = 1$
   (d) $c = 500, n_0 = 1$

3. Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ defined by $f(n) = n^2(n \bmod 2)$ and $g(n) = n^2(n \bmod 3)$. Which of the following is true? (select all that apply)

   (a) $f(n) \in o(g(n))$
   (b) $f(n) \in \Theta(g(n))$
   (c) $f(n) \in \omega(g(n))$
   (d) None of the other answers are true

4. Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ defined by $f(n) = 3n^3 + 10n$ and $g(n) = 10n^3 + 5$. Which of the following is true? (select all that apply)

   (a) $f(n) \in o(g(n))$
   (b) $f(n) \in \Theta(g(n))$
   (c) $f(n) \in \omega(g(n))$
   (d) None of the other answers are true

5. If $f(n) \in \omega(g(n))$, which of the following must also be true? (select all that apply)

   (a) $f(n) \in O(g(n))$
   (b) $f(n) \in \Omega(g(n))$
   (c) $f(n) \in \Theta(g(n))$
   (d) $f(n) \notin O(g(n))$
   (e) $f(n) \notin \Omega(g(n))$
   (f) $f(n) \notin \Theta(g(n))$

4.5 Exercises

1. Prove that $3n^5 + 4n^3 + 12 \in O(n^5)$ using the formal definition of big-O.

2. Prove that $n^3 - 10n - 99 \in \Omega(n^2)$ using the formal definition of big-Ω

3. Let $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$ be functions which output only positive values, and suppose that $f(n) = O(g(n))$. Prove that there exists a constant $c \in \mathbb{R}^+$ such that for all $n \geq 1$, $0 \leq f(n) \leq cg(n)$ (i.e. show that for functions from $\mathbb{Z}^+$ to $\mathbb{R}^+$, we can always choose $n_0 = 1$ in the definition of big-O).

4. (a) Let f_1, f_2, f be functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ such that $f_1(n) = \Theta(f(n))$ and $f_2(n) = \Theta(f(n))$. Prove that $f_1(n) + f_2(n) = \Theta(f(n))$

   (b) Let f be a function from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ and let $f_1, f_2, f_3 \ldots$ be an infinite sequence of functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ such that $\forall\, i \in \mathbb{Z}^+, \quad f_i(n) = \Theta(f(n))$. Define a new function $g(n) := \sum_{i=1}^{n} f_i(n)$. Is it necessarily true that $g(n) = \Theta(nf(n))$? Determine whether or not this is true, and give either a proof or a counterexample.

5 Modeling Computation and Runtime

What does it mean for an algorithm to be efficient? This depends on the context. In general an efficient algorithm is one which doesn't use "too much" of any resource. There are lots of resources that we might care about, but the two that we will care about in this course are *time* (how long the algorithm takes to run) and *space* (how much computer memory the program needs access to).

Comparing the runtime of two algorithms is more difficult than it may sound. In Sections 5.1 and 5.2 we will discuss several choices we need to make in order to meaningfully define what it even means for one algorithm to be faster than another. In Section 5.3, we will discuss a simplified model of computation called the (word) RAM model that allows us to describe algorithms more rigorously than we have done so far and to formally define their runtime. In Section 5.4, we'll show that despite its simplicity, we can actually implement the kinds of features we expect from a modern programming language in the RAM model. In Section 5.5 we'll see an example of a full program in the RAM model and rigorously analyze its runtime. Finally in Section 5.6, we'll show how to use our RAM model-based definition of runtime to analyze algorithms written in pseudocode, like those we've seen in Chapter 3.

5.1 Measuring the Runtime of Algorithms

Given an algorithm, how do we determine how fast it runs? For example, suppose we want to analyze the runtime of the algorithm below (which we previously saw in Section 3.3). How should we do it?

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

One approach would be to implement the algorithm in a real programming language, then run it and time how long the program ran for. This can be a good solution if we know exactly how the algorithm is going to be used. As a general way of comparing algorithms, however, it has serious problems:

- How do we choose what language/implementation to use? (This may affect the runtime significantly)
- What hardware do we run the program on? (The same code can run much faster on some computers than others)
- What input do we run the program on? (Insertion sort will be much faster on an array that is already mostly sorted than on an array that is in reverse sorted order. We need to know what kind of arrays we expect to get as inputs)
- How do we control for other factors that can impact the runtime? (Even on the same input and the same computer, the same program will not always take the same amount of time because of factors like what other processes are using RAM in the background).

Because of these reasons and others, the standard way of comparing the runtime of algorithms is to use a theoretical model. Instead of measuring the real time that a program takes, we measure the number of "steps" that the algorithm performs, where a step is a single basic instruction. On a real computer, these steps will not all take the exact same amount of time, but as long as we make reasonable choices about what to count as a step, the time each one takes on a real computer should be similar. We'll discuss how we determine what exactly counts as a single step of computation in Sections 5.2 and 5.3.

Counting computation steps in an abstract model solves most of the issues mentioned above, but it still leaves us with the problem of choosing what input to run the program on. The truth is that we cannot accurately capture the runtime of an algorithm by running it on any single input. Instead it makes more sense to view the runtime as a function of the input. Assuming that the input to our algorithm is encoded as a single string, we define, for each $s \in \Sigma^*$, Runtime(s) as the number of steps that our algorithm takes on input s.

Typically what we are most interested in is how the *size* (number of characters) of the input affects the runtime. To compute this we need some way to summarize the runtime of the algorithm across all inputs of a given size. Here are three natural ways we could define R(n), the runtime of our algorithm on inputs of size n.

- The *best-case* runtime is defined as $R_{best}(n) := \min_{s \in \Sigma^n}$ Runtime(s)
- The *worst-case* runtime is defined as $R_{worst}(n) = \max_{s \in \Sigma^n}$ Runtime(s)
- The *average-case* runtime is defined as $R_{average}(n) = (1/|\Sigma|^n) \sum_{s \in \Sigma^n}$ Runtime(s)

The best-case is mostly useful for impressing naive investors, but it can occasionally give insights on how to speed up an algorithm (perhaps we can find a way to achieve the best-case runtime on a larger set of strings). The average-case runtime sounds practical, but it is usually complicated to calculate, and assuming that the input is a uniformly random string is often not realistic. The worst-case runtime is what we will focus on in this course. When we talk about the "runtime" of an algorithm, we mean "the worst-case runtime function R_{worst}". An advantage of the worst-case runtime is that it gives us a guaranteed upper bound on how long our algorithm will take (on inputs of size n), even when we know nothing about the specific inputs our algorithm will be given.

We'll calculate the worst-case runtime of InsertionSort later in this chapter, after clarifying a few more details about how we analyze runtimes.

5.2 Defining Steps of Computation

As we discussed in the previous section, we would like to define the runtime of an algorithm on an input as the number of steps/instructions/lines of code that the algorithm executes when run on that input. We have to be careful how we define a "step", however, because different choices can give very different answers for the runtime of the same algorithm.

For example, consider the following high level pseudocode for an algorithm which determines whether or not a given string is a palindrome (reads the same backwards and forwards).

```
IsPalindrome(s):
    Return s = reverse(s)
```

What is the runtime of IsPalindrome? To answer this, let's try to count the number of "steps" that this algorithm takes when we implement it in an actual programming language.

Here is a simple Python implementation of IsPalindrome.

```
def is_palindrome(s):
    return s == s[::-1]
```

It's fine if you aren't familiar with this syntax, all the program is doing is comparing the string s with its reverse and returning whether or not they are equal. In Python, we can do this in a single line of code, regardless of how long s is. So if we define a "step" as "a single line of Python", then this algorithm runs in $\Theta(1)$ time.

If we wanted to write the same algorithm in C (or C++), then we don't have built in syntax for reversing an array, so we would need to write something like

```
bool is_palindrome(char s[], int size) {
    char* reverse = malloc(size);
    for(int i = 0; i < size; i++) {
```

```
        reverse[i] = s[size-i-1];
    }
    for(int i = 0; i < size; i++) {
        if (s[i] != reverse[i]) {
            return false;
        }
    }
    free(reverse);
    return true;
}
```

This version of the function has two for loops, each of which runs for up to `size` iterations, so if we define a "step" as a line of C code, then this algorithm runs in $\Theta(n)$ time.

The first formal models of computation were lambda calculus and Turing machines, developed in the 1930s by Alonzo Church and his student, Alan Turing. Defining these models is outside the scope of this course, but it can be proven that they have the same power as modern programming languages in the sense that any function which can be computed by a Python program (for example) can also be computed by some Turing machine. This does not mean, however, that a Turing machine can compute any function with the same asymptotic efficiency as a Python program.

A key distinction between Turing machine and actual programming languages like C is that Turing machines do not have random access, i.e. a Turing machine has no equivalent to the expression `s[i]` in our C program. Instead, a Turing machine has a single "head", which moves through memory one symbol at a time. Thus, if the Turing machine takes an array s as input and wants to access $s[i]$, it has to "move right" i times (i.e. the Turing machine treats an array more like a doubly linked list). If we were to implement the `is_palindrome` on a Turing machine it would look something like this (once again, it is not important to understand the details of this program):

```
is_palindrome(s):
    while head is not blank:
        char c = head
        write blank
        move right
        if head is blank:
            accept
        while head is not blank:
            move right
        move left
        if head != c:
            reject
        write blank
        move left
        while head is not blank:
            move left
        move right
    accept
```

This program features nested while loops. The outer loop iterates through $\lfloor n/2 \rfloor$ pairs of characters, and for each pair the inner while loops have to travel all the way across the input array and then all the way back to check if the first character of the input matches the last one. Thus, if we define a "step" as a single Turing machine instruction, then IsPalindrome runs in $\Theta(n^2)$ time.

So which of these is the right answer? Does IsPalindrome run in $\Theta(1)$, $\Theta(n)$, or $\Theta(n^2)$ time? A good way to build intuition here is to try running the Python and C programs that we wrote on an actual computer. While we said before that "the time the algorithm takes on an actual computer" is too imprecise to be a definition of runtime, we still want the asymptotic runtimes of programs in our theoretical model to match their asymptotic runtimes on actual hardware (i.e. if the number of

seconds that a program takes to run grows quadratically with the input size, ideally it should be an $O(n^2)$ time algorithm in our model).

If we run both the Python and C programs on strings of various sizes we find that the runtime for both programs seems to be linear in the length of the input string, suggesting that $O(n)$ is the right answer for this algorithm. In fact, if we look at how the Python interpreter translates the line `return s == s[::-1]` into machine code, we'll find that it uses a loop structure similar to our C program.

This doesn't mean that "a line of C code" is the right definition of a step of computation, however. We can still write single lines of C code that take a long time to run (e.g. if `a` and `b` are structs, then `a=b` copies all the attributes of `b` into `a`, which may take a long time if the struct is large). Furthermore, even if we did find a programming language where each instruction takes about the same amount of time (maybe some version of assembly), real programming languages are far too complicated, with too many instructions and edge cases to make them a reasonable definition of computation (the C++ standard, for example, is over 1000 pages long!). We want a simple minimalist model (like a Turing machine, which can be described in 1 page), but one which is powerful enough to allow manipulating arrays/strings with the same asymptotic efficiency as C or Python. It turns out that the right answer is essentially a Turing machine augmented with the power of random access, i.e. a "Random Access Machine".

5.3 The RAM Model

The property of being able to jump to any desired location in an array (or more generally, in memory) just using the index is called "random access". Essentially all modern computers rely heavily on RAM (random-access memory) so to model the time that operations take on real computers, it makes sense to use a model which includes random access. You can think of the RAM model as a very simple programming language.

Every RAM model program takes a string as input and returns a string as output (i.e. computes a function from $\Sigma^* \to \Sigma^*$ for some alphabet Σ). This is not actually a major limitation, because any type of input or output that a regular program uses is ultimately represented using a string of bytes. We'll talk more about encoding objects as strings in Section 6.1.

Memory in the RAM model consists of one long array of cells, which we call M. Each cell of M stores a single character from Σ. Initially the size of M is equal to n, the input length, and M stores the input to the algorithm in cells $M[1]$ through $M[n]$. While running, the machine can increase the size of M by using the "Malloc" instruction, which adds one cell to the end of M containing a specified character.

In addition to the memory array (which can grow arbitrarily large) a RAM model program also has a finite number of variables. RAM model programs have only two types of variables, character variables, which store a single symbol from Σ and integer variables. Integer variables in the RAM model function like a cross between int and pointer types in a language like C, because we can perform arithmetic with them, but we can also use them to index into memory. Like integer and pointer variables in C or many other languages, integer variables in the RAM model cannot store arbitrarily large integers. Instead, integer variables can store values in the range $[-2^w, 2^w]$, where w is a value called the "word size".

On a real computer, the word size is a fixed constant. These days most computers have a word size of 64 (i.e. a 64-bit operating system), but in the 90s, a word size of 32 was most common, and in the 70s many computers used a word size of 16. Because the word size bounds the values of variables which are used as indices into memory (i.e. addresses), the word size for a particular computer needs to be large enough to store all of the possible indices of memory locations. So if our computer has word size w, the maximum amount of RAM that it can use is 2^w. This means that with a word size of 16, we can use up to 64 kilobytes of RAM, with a word size of 32, we can use up to 4 gigabytes, and with a word size of 64, we can use up to 16 exabytes (about 17 billion gigabytes). This is why the word size used in actual computers has increased over time, because the amount of memory that we can cram into a computer chip has increased exponentially over the past several decades.

In our model, using a fixed word size doesn't work. We want to be able to describe algorithms that work for arbitrarily large inputs (because we are interested in the asymptotic behavior of their runtime), and so our machine's memory has unbounded size. Therefore, we define the word size w as $\max(64, \lceil \log_2(\text{memory\_size}+1) \rceil)$, so that we always have $\text{memory\_size} < 2^w$, meaning that we can store the index of any memory cell in an integer variable, but we also have $w \geq 64$ so that we store the kind of values that fit in an integer variable on a modern computer even if our program's memory is very small (e.g. if our program takes an empty string as input, then memory is initially empty). Note that this means that the word size can change over the course of the program as we allocate more memory.

Formally, a program in the (word)-RAM model consists of:

- An alphabet Σ
- A finite collection of variables (sometimes called "registers"), each of which is either a character variable (storing a single character) or an integer variable (storing a single integer between -2^w and 2^w, inclusive) including the following special variables which cannot be assigned to directly, only modified by the Malloc instruction):
  - memory_size: an integer variable initialized with the number of characters in the input
  - word_size: an integer variable initialized with $\max(64, \lceil \log_2(\text{memory\_size}+1) \rceil)$
- A finite list of instructions (or "lines of code") where each instruction is one of the following:
  - $a \leftarrow b$
    - a must be a variable and b must be a variable or constant of the same type (integer or character).
    - This instruction copies the value of b into the variable a.
  - $M[i] \leftarrow c$
    - i must be an integer variable or constant and c must be a character variable or constant.
    - This instruction copies the value of c into position i in memory. If i is out of bounds, no memory is changed.
  - $c \leftarrow M[i]$
    - i must be an integer variable or constant and c must be a character variable.
    - This instruction copies the character at position i of memory into c. If i is out of bounds, c is left unchanged.
  - $i \leftarrow j$ op k
    - i must be an integer variable, j and k must be integer variables or constants, and op must be one of the following: $+, -, *, /$.
    - This instruction stores the result of j op k in i. / performs integer division (i.e. the fractional part is discarded). If j op k would not fit in an integer variable (because it is $< -2^w$ or $> 2^w$) or is undefined (because of division by 0), i is left unchanged.
  - If a op b: goto ℓ
    - a must be a variable, b must be a variable or constant of the same type (integer or character) as a and ℓ must be the number of an instruction (i.e. an integer between 1 and the total number of instructions), and op must be one of the following: $<, \leq, =, \geq, >$.
    - If the comparison a op b is true, then this instruction causes the machine to jump to instruction number ℓ and continue executing instructions from that position. If the comparison is false, the machine proceeds to the instruction immediately after this one, as usual.

- $i \leftarrow \text{decode}(start, size)$
  - i must be an integer variable, while $start$ and $size$ must be integer variables or constants.
  - This instruction interprets the string of length $size$ symbols beginning at position $start$ in memory (i.e. $M[start : start + size - 1]$) as the representation of a single integer, and stores that integer in i. If these characters do not represent a valid integer (either they don't encode an integer or the encoded integer is $< -2^w$ or $> 2^w$) then i is left unchanged.
- $M[start, size] \leftarrow \text{encode}(i)$
  - $i, start$, and $size$ must be integer variables or constants.
  - This instruction encodes i as a string over Σ and stores the resulting string beginning at index $start$. If the encoding of i has fewer than $size$ characters, it is padded with leading 0s until it has length $size$ (this is analogous to how an integer in C++ takes up 4 bytes, whether the number being stored is 1 or 100000). If the encoding of i has more than $size$ characters, then memory is left unchanged.
- $\text{Malloc}(c)$
  - c must be a character variable or constant.
  - This instruction adds a single new cell storing c to the end of M. It also increases memory_size by 1. If this causes the length of M to become 2^w, then this instruction also increases w by 1 (meaning that integer variables can now store larger values, so that it is possible to index into this new memory).
- $\text{Output}(start, size)$
  - $start$ and $size$ must be integer variables or constants.
  - Ends the program, outputting the string of length $size$ beginning at $M[start]$ (i.e. $M[start], \ldots, M[start + size - 1]$).

Since the RAM model is not a real programming language, different authors have their own slightly different versions of it. We could've allowed more instructions, or slightly fewer, without changing the asymptotic runtime of programs in the model (we'll see in Section 5.4 how some features we didn't define as part of the model, like loops and functions, are actually pretty easy to implement).

The one major difference you may encounter relates to the word size. The earliest descriptions of the RAM model did not feature a word size: variables could store arbitrarily large integers. This leads to an unrealistic model, however, because real computers cannot perform arithmetic on arbitrarily large integers in constant time. In fact, the time to perform arithmetic operations is a major bottleneck in areas like cryptography, where working with numbers with thousands of digits is common. Therefore the version of the RAM model where the values for integer variables are limited by the word size (often called the word-RAM model to distinguish it from the version without a word size) is the one most commonly used in modern algorithms research, and is the only one we will care about in this course. We'll discuss the runtime of arithmetic with large integers in our model in Chapter 9.

5.4 Implementing Quality of Life Features in the RAM Model

The instruction set we gave in Section 5.3 may seem pretty restrictive. It lacks a lot of nearly universal programming language features like loops, functions, and arrays. It turns out, however, that even with the limited instruction set of the RAM model, we can simulate all of these features and more. In this section we'll give some examples of how to do this.

First let's look at implementing if statements. An if-else statement like the one below (where "do X" is a stand-in for additional blocks of code) can be implemented in the RAM model using our conditional goto.

```
if x = y:
    do A
else:
    do B
do C
```

An equivalent RAM model program is:

1: If $x = y$: goto 4
2: do B
3: If $0 = 0$: goto 5
4: do A
5: do C

Note that we used the always true statement $0 = 0$ to convert the conditional goto into a statement that always jumps to the given line, so that we can avoid executing the "if" and "else" cases together. Also note that because our program moves on to the next line after the if when the condition is not met, we put the code from the else block before the code from the if block.

Our translation of an if-else statement relied on the condition in the if statement having a simple form like "x = y", because our conditional goto instruction requires the condition to be a basic comparison between two variables. We can simulate more complicated boolean expressions in the RAM model, however. For example, we can translate

```
if w = x and y = z:
    do A
do B
```

into

1: w_equals_x $\leftarrow 1$
2: If $w = x$: goto 4
3: w_equals_x $\leftarrow 0$
4: y_equals_z $\leftarrow 1$
5: If $y = z$: goto 7
6: y_equals_z $\leftarrow 0$
7: true_count $\leftarrow$ w_equals_x + y_equals_z
8: If true_count < 2: goto 10
9: do A
10: do B

Here we have created new "boolean" variables (integer variables that we only set to 0 or 1) to represent the truth values of the two sides of the "and". To check if both are true, we add up the truth values and check if the total is 2. We can implement other logical operators in a similar way (e.g. an "or" statement is true if the sum of the truth values of both sides is at least 1).

Now that we know how to implement a variety of conditionals, we can also implement loops. A while loop like

```
while x ≤ y:
    do A
do B
```

becomes

1: If $0 = 0$: goto 3
2: do A
3: If $x \leq y$: goto 2
4: do B

and a for loop like

```
for i from 1 to n:
    do A
do B
```

becomes

1: $i \leftarrow 1$
2: If $i > n$: goto 6
3: do A
4: $i \leftarrow i + 1$
5: If $0 = 0$: goto 2
6: do B

Conditional goto allows us to repeatedly return to the beginning of the loop as long as the loop condition is met.

Our RAM model has only two primitive data types, characters and integers, but we can create more complicated data types in memory out of these basic building blocks. For example, if we want to store an array of characters (i.e. a string), we can use two variables, one to store the index where the array starts, and one to store the length of the array. For example, we could represent a string s with the variables s_{start} and s_{length}. Then, to access $s[i]$, we simply need to access $M[s_{start} + i - 1]$ (since our indices for arrays and strings begin at 1).

We can represent an array of integers in a similar way, however, if we want to allow the integers in the array to be arbitrarily large, we should add a third variable keeping track of the number of characters we're using to represent each integer. Given variables A_{start}, A_{length}, and $A_{element\_size}$, we can access $A[i]$ via $\text{decode}(A_{start} + (i-1) * A_{element\_size}, A_{element\_size})$.

We can also implement function calls in our RAM model, although the process is a bit more complicated (akin to how function calls are actually translated into low-level machine code, which also relies on goto instructions).

- We can jump to the code of a function with an unconditional goto (if $0 = 0$ goto)
- In order to return from the function, however, we need a way to remember which line of code called the function.
- Since we can have arbitrarily long chains of functions calling each other, we need to store this line number to return to in memory.
- Similarly, the parameters/local variables of the function also need to be stored in our memory array M
  - We could try making the parameters global variables (the only kind we have in the RAM model) instead, but this approach cannot handle recursive functions, which can have multiple different copies of their local variables existing at the same time.
- In essence what we want is an array of "stack frames" where each frame stores the local variables for a function as well as the instruction to return to when the function finishes.
- The details of this process are too technical for this course, but hopefully this gives you a sense of how we could translate large complicated programs into the RAM model

There are many more features we could implement, but this should be enough to give you some idea of how we can reduce more complicated code down to the instruction set from the RAM model. It turns out that in fact any function mapping strings to strings which can be computed by a program in C or Python or another normal programming language can also be computed by a program in the RAM model. Proving this is rather tedious, however, and beyond the scope of this course.

5.5 Analyzing the Runtime of Algorithms

Now that we've fully described a computational model, we can finally define the runtime of an algorithm. The (worst-case) runtime of an algorithm ALG is a function $f : \mathbb{Z}_{\geq 0} \to \mathbb{Z}^+$ where $f(n) :=$ the maximum over all $x \in \Sigma^n$ of the number of RAM model instructions that ALG(x) executes.

Let's look at a couple of examples of full programs in the RAM model and analyze their runtimes. We'll start with the IsPalindrome algorithm that we discussed in Section 5.2.

```
IsPalindrome(s):
  Return s = reverse(s)
```

Here is an implementation of IsPalindrome in our model. Lines 1 through 9 compute the reverse of the input string, storing it in positions $n+1$ through $2n$ in memory, and then lines 10 through 32 compare the string with its reverse to check for equality. As you can see, even an algorithm that is extremely conceptually simple can have a rather long description in the RAM model!

- Alphabet: Any set containing all possible input characters as well as English letters a-z.
- Integer variables: memory_size, word_size, input_size, i, j
- Character variables: temp, forward_char, reverse_char

1: input_size ← memory_size
2: i ← 0
3: If i = input_size: goto 9
4: j ← input_size - i
5: temp ← $M[j]$
6: Malloc(temp)
7: i ← i + 1
8: If 0 = 0: goto 3
9: i ← 1
10: If i > input_size: goto 25
11: forward_char ← $M[i]$
12: j ← i + input_size
13: reverse_char ← $M[j]$
14: i ← i + 1
15: If forward_char = reverse_char: goto 10
16: If memory_size ≥ 5: goto 19
17: Malloc('a')
18: If 0 = 0: goto 16
19: $M[1]$ ← 'f'
20: $M[2]$ ← 'a'
21: $M[3]$ ← 'l'
22: $M[4]$ ← 's'
23: $M[5]$ ← 'e'
24: Output(1,5)
25: If memory_size ≥ 4: goto 28
26: Malloc('a')
27: If 0 = 0: goto 25
28: $M[1]$ ← 't'
29: $M[2]$ ← 'r'
30: $M[3]$ ← 'u'
31: $M[4]$ ← 'e'
32: Output(1,4)

Alright, so what is the runtime of this algorithm?

Theorem 27. *The runtime of* IsPalindrome *is* $12n + 11 + 3\max(0, 4 - 2n)$.

Proof. We break the program into pieces.

- Lines 1-2 run once each, so the total time (number of instructions run) for these lines is just 2.
- Lines 4-8 each run input_size many times (since i starts at 0 and they run until i is equal to input_size. Line 3 runs input_size + 1 times (since it is also executed in the final iteration when i is equal to input_size). Thus the total time for lines 3-8 is $6n + 1$.

- Line 9 runs once, for a total time of 1.
- The runtime of the rest of the program depends on the actual characters in the input—in particular it depends on whether or not the input string is a palindrome. Let's separately consider both cases
- First suppose that the input string is a palindrome
  - Since the input is a palindrome, forward_char will always be equal to reverse_char on line 15. Thus, lines 11-15 run input_size many times (once for each character of the input), and line 10 runs input_size + 1 times (since it also runs in the final iteration when i is equal to input_size + 1). Thus, the total time for lines 10-15 is $6n + 1$
  - After matching all of the characters in the forward and reverse strings, the program jumps to line 25, so lines 16-24 do not run at all.
  - When we reach line 25, memory_size is equal to double input_size (since we doubled the size of memory to create the reversed copy). Therefore lines 26-27 run $\max(0, 4 - 2n)$ times and line 25 runs $1 + \max(0, 4 - 2n)$ times (since it runs each time our memory size is less than 4, plus one extra time when our memory size is at least 4). Thus, the total time for lines 25-27 is $1 + 3\max(0, 4 - 2n)$.
  - Lines 28-32 run once each, for a total time of 5
  - Thus, if the input is a palindrome the total runtime of IsPalindrome is

$$2 + (6n + 1) + 1 + (6n + 1) + 1 + 3\max(0, 4 - 2n) + 5 = 12n + 11 + 3\max(0, 4 - 2n)$$

- Now suppose that the input x is not a palindrome. In this case, there must be some index i where $x[i] \neq \text{reverse}(x)[i]$. Furthermore, the first index where this happens must be less than or equal to $n/2$ (i.e. before the middle of the string), because when we are comparing x to its reverse, every character in the first half gets paired with one from the second half and vice-versa, so every unmatched pair has a character from the first half of the string.
- Therefore, lines 10-15 each run at most $\lfloor n/2 \rfloor$ times before we reach the non-matching pair of characters and continue on to line 16 (and they run exactly this many times if the input is an "almost palindrome" where only the middle pair of characters don't match), so the worst-case runtime lines 10-15 is $6\lfloor n/2 \rfloor$
- As above, we note that the current memory_size is $2n$, so lines 16-18 run $1 + 3\max(0, 5 - 2n)$ times in total.
- Lines 19-24 run once each, for a total time of 6
- line 24 ends the program, so lines 25-32 never run
- Thus, the worst-case total runtime when the input is not a palindrome is

$$2 + (6n + 1) + 1 + 6\lfloor n/2 \rfloor + 1 + 3\max(0, 5 - 2n) + 6 = 6n + 6\lfloor n/2 \rfloor + 11 + 3\max(0, 5 - 2n)$$

The overall worst-case runtime is the maximum of the worst-case runtime for palindromes and the worst-case runtime for non-palindromes:

$$\max(12n + 11 + 3\max(0, 4 - 2n), 6n + 6\lfloor n/2 \rfloor + 11 + 3\max(0, 5 - 2n)).$$

We note that if $n \geq 3$, then $\max(0, 4 - 2n) = \max(0, 5 - 2n) = 0$, and so the runtime on palindromes is strictly larger than the runtime on non-palindromes. We can also check that for $n = 2$, the runtime for palindromes is still larger (the extra iterations of the loop comparing characters takes longer than the extra time to output false). Finally, if $n \leq 1$, the input must be a palindrome (an empty string

or a single character string is always the same as its reverse). Thus, for all input lengths the worst case input is a palindrome, and we can simplify the runtime to:

$$12n + 11 + 3\max(0, 4 - 2n).$$

□

That was an insane amount of effort that we would like to never go through again. Fortunately, calculating the *exact* runtime of algorithms in our model is not important (it depends too much on the specific choice of instruction set for the RAM model, which, as we previously noted is somewhat arbitrary). Instead what we really care about is the asymptotic runtime of the algorithm. Using the tools we learned in Chapter 4, we can confirm that the runtime expression we obtained for IsPalindrome is $\Theta(n)$. As we'll see in the next section, however, as long as all we care about is this final asymptotic answer, we can avoid needing to write out the fully messy RAM model program and calculating the exact runtime function.

5.6 Pseudocode

The simplicity of the RAM model is useful as a definition for runtime. We can calculate the exact number of steps needed for any operation that our program uses by breaking it down into individual RAM model instructions. Writing anything but the simplest algorithms directly in the RAM model is rather tedious, however, and in this course we are not going to concern ourselves with the exact number of steps that an algorithm takes.

Therefore, for the rest of this course, instead of describing algorithms using only RAM model instructions, we will write them in higher level pseudocode, like what we used in Chapter 3 when we discussed proving the correctness of algorithms.

Pseudocode does not have a specific list of valid instructions. In a similar way to how proofs can be written in regular English, using mathematical notation only when it is the clearest way to convey an idea, you can use everyday language in pseudocode instead of trying to adhere to a strict set of syntax rules. I will attempt to use a consistent style for the pseudocode algorithms in these notes, but you do not have to follow it when describing your own algorithms. If you prefer to make your pseudocode look more like your favorite programming language, that's fine, as long as the intent would be clear to someone who doesn't know that language.

A heuristic for whether something is a valid "single instruction" in pseudocode is whether you (or a slightly less clever classmate) would be able to break it down into the basic RAM model instructions. For example, in the previous section, we saw how to write a loop in the RAM model that reverses a string. So it's fine to write pseudocode like the following (which we've seen a bunch of times at this point), where we treat "reverse" as a built in operation, as long as we remember that this instruction actually takes $O(|s|)$ time.

```
IsPalindrome(s):
    Return s = reverse(s)
```

To analyze the runtime of an algorithm written in pseudocode, we don't need to convert each line into RAM model instructions. Instead all we need to figure out is the asymptotic behavior of the RAM model equivalent (e.g. could we implement this in a constant number of RAM model instructions). Generally it will be clear which lines run in constant time, and you can state this without justification (i.e. you don't need to describe the RAM model program at all). Be careful not to make the assumption that every non-loop line of pseudocode corresponds to a constant number of RAM model instructions, however. As we've seen with "reverse" this isn't always the case. We'll see more examples in chapter 9 of pseudocode lines that may look constant time at first glance but actually require many RAM model instructions.

Let's see an example of analyzing the asymptotic runtime of a pseudocode algorithm.

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Theorem 28. *The runtime of* InsertionSort *is* $\Theta(n^2)$.

Proof. First we show that the runtime is $O(n^2)$. We note that the outer loop runs n times, and the inner loop runs at most n times, so the body of the inner loop runs at most n^2 times. The body of the loop takes $O(1)$ time per iteration, so the overall runtime is $O(n^2)$.

To show that the runtime is $\Theta(n^2)$, however, we also need to show that the worst-case runtime is $\Omega(n^2)$, i.e. we need to show that for every sufficiently large n we can find an input on which InsertionSort actually takes $\Omega(n^2)$ time.

Suppose that A is in reverse sorted order (i.e. sorted from largest to smallest). Then at the start of every iteration of the outer for loop, $A[i]$ will be smaller than every element of $A[1 : i-1]$, so the inner for loop will run the full $i-1$ times. Each iteration of the inner for loop includes at least one instruction, so we can lower bound the total runtime with the total number of iterations of the inner for loop.

Summing across all n iterations of the outer for loop, we find that the total number of iterations of the inner for loop is

$$\sum_{i=1}^{n}(i-1) = \sum_{i=1}^{n-1} i = (n-1)n/2 = n^2/2 - n/2 \in \Omega(n^2).$$

Note that we simplified the summation using Theorem 3. □

If you are paying very close attention, you may notice that this proof cheated. Look back and see if you can find the false claim.

With what we've stated so far, we can't assume that the body of the loop runs in $O(1)$ time! While comparing two integer variables is a constant time operation in the RAM model, remember that integer variables in the RAM model can store values at most 2^w, where $w = \max(64, \lceil \log_2(\text{memory\_size}+1) \rceil)$. Thus, if our array A contains very large numbers, we may not be able to compare them in constant time. Consider, for example, the case where A contains just two numbers, each with millions of digits.

If A actually contains incredibly huge numbers, however, then n is not actually a good measure of the input size. e.g. if A has two numbers with millions of digits, then input_size is "millions", but n is only 2. It is much more convenient to work with the number of elements in an array rather than the total number of characters/bits. Therefore, unless otherwise stated, whenever we have an algorithm which takes an array of integers as input, we will implicitly assume that each of the elements of the array has at most $100w$ digits (there's nothing special about 100 here, it's just a nice large constant). That way we guarantee that we can store each one in a constant number of RAM model variables and perform comparisons (and arithmetic) on them in constant time. Given this new assumption, the proof of Theorem 28 is correct.

5.7 Comprehension Questions

1. When we talk about the "runtime" of an algorithm in this course, which runtime function do we mean?
   (a) The best-case runtime
   (b) The average-case runtime
   (c) The worst-case runtime

2. True or false: in the RAM model, the word size is a fixed constant which cannot change during the execution of a program.

3. Suppose that we run a RAM model program with the empty string as input. At the start of that program, what is the largest value that it can store in an integer variable?

4. Which of the following are valid single instructions in a RAM model program? (assume that a, b, and c are integer variables).

   (a) $M[a] \leftarrow b + c$

   (b) If $a \bmod 2 = 1$: goto 1

   (c) $a \leftarrow \text{decode}(b, c)$

   (d) $a \leftarrow b/c$

   (e) Malloc

   (f) Output(a, c)

5. Give a big-Θ expression for the runtime of IsPalindrome.

6. Give a big-Θ expression for the runtime of InsertionSort.

7. Give an exact expression (not big-O/big-Θ) for the runtime of each of the following RAM model programs in terms of the input size n:

   1: If memory_size ≥ 10: goto 4
   2: Malloc('0')
   3: If $0 = 0$: goto 1
   4: Output(1,1)

8. Below is an algorithm which takes as input a non-empty array of n distinct integers and outputs the median. Give a simple (polynomial) big-Θ expression for the runtime of the algorithm.

```
FindMedian(A[1, ..., n]):
    While |A| > 2:
        x ← max(A)
        y ← min(A)
        remove x from A
        remove y from A
    If |A| = 1:
        Return A[1]
    Else:
        Return (A[1] + A[2])/2
```

9. Below is an algorithm which takes as input a string of length n and reverses it. Give a simple (polynomial) big-Θ expression for the runtime of the algorithm.

```
Reverse(s):
    n ← |s|
    For i from 1 to n:
        temp ← s[n]
        For j from n down to i + 1:
            s[j] ← s[j − 1]
        s[i] ← temp
    Return s
```

5.8 Exercises

1. The RAM model does not have a "mod" operation. Show that nevertheless we can implement $a \leftarrow b \mod c$ in a constant number of RAM model instructions. That is, give a sequence of RAM model instructions which will set a to $b \mod c$, assuming that a, b, c are existing integer variables. You may assume that b and c both have **positive** values. You may introduce any additional variables you wish.

2. Translate this pseudocode into a RAM model program.

```
CountAs(s):
    c ← 0
    For i from 1 to |s|:
        If s[i] = 'a':
            c ← c + 1
    Return c
```

3. In Section 5.6 we claim that we can represent numbers with at most $100w$ digits in a constant number of RAM model variables and perform arithmetic on them in constant time. Justify this by explaining how to actually represent numbers with $100w$ binary digits and perform addition on them in the RAM model (Hint: start by showing how to represent and manipulate numbers with $2w$ binary digits).

4. Determine with proof the runtime of this algorithm

```
ExchangeSort(A[1, ..., n])
    Repeat forever:
        i ← −1
        For j from 1 to n − 1:
            If A[j] > A[j + 1]:
                i ← j
        If i = −1:
            //A is now sorted
            Return
        Swap A[i] with A[i + 1]
```

6 Decidability

In Chapter 5 we saw how to formally define an algorithm as a program in the RAM model, and gave some intuition for why this definition allows us to describe any algorithm that we can write on an actual computer.

In this chapter we will explore the topic of computability. In Section 6.1 we discuss how we can encode objects as strings so that we can create algorithms with inputs or outputs of whatever type we want, including algorithms which take other algorithms as input. In Section 6.2 we will define decidability and look at how to prove that a language is decidable. Finally, in Section 6.3 we will see our first example of a problem that algorithms cannot solve.

6.1 String Encodings and Universal Machines

A RAM model program takes as input a single string. This may seem limiting, since many functions we care about involve inputs other than strings (such as integers, arrays, or graphs). It turns out, however, that we don't lose much by restricting ourselves to string inputs, because all kinds of other objects can be *encoded* as strings. For some object A, we use $\langle A \rangle$ to denote the string representation of A. Similarly for a sequence of objects $A_1, \ldots, A_k$ we use $\langle A_1, \ldots, A_k \rangle$ to denote a single string encoding all of the objects.

Note that there are a few objects we care about which cannot be encoded as strings. In particular, we cannot encode arbitrary real numbers as strings. Intuitively, this is because real numbers can have infinitely long decimal representations with no pattern to the digits, and our algorithms can only take *finite* input strings. Thus, our algorithms cannot take arbitrary real numbers as inputs or outputs, only ones which can be expressed as finite strings (e.g. rational numbers, which can be encoded using their numerator and denominator, even if their decimal representations would be infinitely long).

There are many different ways to encode the same object as a string (for example, we could represent integers in binary or in base-10 or in some more exotic system), and in future chapters it will sometimes be important to specify which one we're using. For this chapter, however, we are only focused on the correctness of our algorithms/programs, not their runtime or space usage, so the particular encoding scheme that we use will generally not matter. Therefore we will not add unnecessary clutter by formally specifying what encoding schemes we are using.

Whenever we write that an algorithm takes a particular kind of object as input, we implicitly mean that the algorithm interprets its input as an encoding of that object (i.e. if we say that an algorithm takes an integer as input, the actual input is the string $\langle n \rangle$, but the first thing that our algorithm does is convert this string into an integer).

When we say that an algorithm returns a particular object, we mean that the algorithm outputs the encoding of that object. For example, when working with the alphabet $\Sigma = \{0, 1\}$, the instruction "Return True" may actually be implemented by outputting the string "1", but we will not get into these encoding details in our pseudocode.

One important class of objects which we can encode with strings is algorithms themselves. After all, the source code of a program in the RAM model (or any programming language) is just a long string. This means that we can write algorithms that take the encodings of other algorithms as input (you probably use such a program frequently—one example is a compiler). Just as for other objects, we will use $\langle P \rangle$ to represent the encoding of the algorithm P as a string (we will adopt the convention of naming generic algorithms with P for "program" to avoid confusion with arrays which we often call A).

A **universal machine** is an algorithm that takes as input the encoding of an algorithm and a string, and simulates running the algorithm on the string (You can think of a universal machine as the RAM model's equivalent of the Python interpreter). More formally, a universal machine takes as input a string in the form $\langle P, x \rangle$, where P is an algorithm and x is a string. If $P(x)$ outputs a string, then the universal machine outputs the same string. If $P(x)$ loops forever (never reaching an "Output" instruction) then the universal machine also loops forever when run on $\langle P, x \rangle$.

It's not obvious that a universal machine should exist. Describing how to construct a universal machine was one of the major contributions of Alan Turing's paper "On Computable Numbers, with an Application to the Entscheidungsproblem", which is also where he first defined Turing machines. At a very high level, our universal machine can use an integer variable to keep track of which line of P it is currently simulating, and designate one section of its memory for storing the code of P, another section to store the memory of P, and a final section for scratch space used to help with tasks like converting the memory addresses used by P into addresses in the universal machine's memory. Formally describing a universal machine is tedious and not the focus of this course, however, so we omit the details. For the purposes of algorithms you write in this course, you may assume that a universal machine exists and therefore that algorithms which you write can simulate the execution of an algorithm given its encoding.

It's important to note that while simulating P with a universal machine preserves the output, it does not necessarily preserve the runtime. Running a universal machine on $\langle P, x\rangle$ will generally be asymptotically slower than directly running $P(x)$. Nevertheless, since we are not concerned with runtime in this chapter, we will allow ourselves to treat an algorithm which we have read from its source code like a regular function we have written and include in our pseudocode lines like "result $\leftarrow P(x)$" as an abbreviation for "Simulate running $P(x)$ and store the output in result".

6.2 Decidability

For the rest of this chapter, we will focus on algorithms that return only True or False.

We define a **language** as a set of strings. That is, for a fixed alphabet Σ, a language is a subset of Σ^*. For example, if our alphabet is $\{0, 1\}$, then $\{010, 1111, \varepsilon\}$ is a language of size 3, and $\{1^n 0 \mid n \in \mathbb{Z}^+\}$ is a language of infinite size. For an algorithm P, we define the **language of** P, denoted $L(P)$ by

$$L(P) := \{x \in \Sigma^* \mid P(x) \text{ returns True}\}.$$

If L is a language and P is an algorithm with $L(P) = L$, we say that P **accepts** L. Note that P may return False or loop on strings not in L. We say that a language L is **acceptable** or **semi-decidable** if there exists an algorithm that accepts L.

If L is a language, P is an algorithm that accepts L, and P returns False on all strings in $\Sigma^* \setminus L$, we say that P **decides** L. We say that a language L is **decidable** if there exists some algorithm that decides L. Note that every decidable language is decided by infinitely many different algorithms (we can always add extra pointless lines of code, e.g. increment a variable then decrement it). An algorithm which always returns True or False (never gets stuck in an infinite loop) is called a **decider**. Note that if P is a decider then P decides $L(P)$, while if P is not a decider then P merely accepts $L(P)$.

The simplest way to prove that a language is decidable is to give an algorithm and prove that the algorithm decides the language.

Theorem 29. *Let*

$$L := \{\langle n, p\rangle \mid n, p \in \mathbb{Z}^+, \text{ and there are at least } p \text{ prime numbers between 1 and } n\}$$

Show that L is decidable.

Proof. We show that L is decidable by giving an algorithm to decide it.

```
L-Decider(n, p)
    count ← 0
    For i from 2 to n:
        prime ← True
        For j from 2 to i − 1:
            If i mod j = 0:
                prime ← False
        If prime:
            count ← count+1
    If count ≥ p:
        Return True
    Return False
```

We note that, by definition, a positive integer $i \geq 2$ is prime if and only if it is not divisible by any positive integer between 2 and $i-1$, or equivalently if $i \mod j \neq 0$ for every j between 2 and $i-1$. Therefore, after the outer for loop of L-Decider the variable count stores the number of primes between 2 and n. This is the same as the number of primes between 1 and n, since 1 is not prime. Since L-Decider returns True if and only if count $\geq p$, L-Decider returns True if $\langle n, p \rangle \in L$ and returns False otherwise. Thus, L-Decider decides L. □

There are several things to note about this proof:

- The definition of a decidable language is one with an algorithm which decides it. Therefore to prove that a language is decidable, we can give an algorithm and then prove that this algorithm correctly decides the language.
- It's not sufficient to simply give the algorithm. We need to be able to prove that our algorithm actually does what we claim it does.
- Although we formally define an algorithm as a RAM model program, to prove that an algorithm exists it is sufficient to give high level pseudocode, since as we discussed in Section 5.6, any feature we can implement in a regular programming language we can also implement in the RAM model.
- Recall that an algorithm always takes a string as input. When we describe an algorithm as taking some other kind of input, we implicitly assume that the algorithm returns False when given a string that does not encode an object of this form. For example, in our proof we assumed that L-Decider takes as input a pair of positive integers, n and p (because this is what elements of L encode). What we mean by this is that L-Decider will begin by checking if its input encodes a pair of positive integers and return False if it does not.
- Any algorithm which decides a language is sufficient to prove that the language is decidable. It doesn't have to be a fast or clever algorithm (we also don't have to actually calculate the runtime of the algorithm). Often, the easiest way to prove a problem is decidable is some kind of "brute force" algorithm. There are much faster algorithms for this primality problem, but using one of them would just be making our correctness proof harder.

Theorem 30. *The language*

$$\{\langle P, x, k \rangle \mid P \text{ is an algorithm}, x \in \Sigma^*, k \in \mathbb{Z}^+, \text{ and } P(x) \text{ halts after at most } k \text{ steps}\}$$

is decidable.

Proof. Call this language L. We show that L is decidable by giving an algorithm to decide it.

```
L-Decider(P, x, k)
    Simulate P(x) for k steps
    If P(x) halted within the simulation:
        Return True
    Return False
```

We note that simulating $P(x)$ for only k steps can be done in finite time, even if $P(x)$ loops forever. To prove that L-DECIDER correctly decides L we need to show that $\forall\langle P, x, k\rangle \in L$, L-DECIDER$(P, x, k)$ returns True and $\forall\langle P, x, k\rangle \in \Sigma^* \setminus L$, L-DECIDER$(P, x, k)$ returns False. In this case both directions are straightforward. Let P be an algorithm, x be a string, and k be a positive integer.

- Suppose $\langle P, x, k\rangle \in L$. Then, by the definition of L, $P(x)$ halts after at most k steps. Therefore, when L-DECIDER simulates $P(x)$ for k steps, $P(x)$ will halt within the simulation, so L-DECIDER(P, x, k) will return True.
- Now suppose $\langle P, x, k\rangle \in \Sigma^* \setminus L$. Then by the definition of L, $P(x)$ is still running after k steps. Therefore, when L-DECIDER simulates $P(x)$ for k steps, $P(x)$ will not halt, and so L-DECIDER$(\langle P, x, k\rangle)$ will return False.

□

In this proof our algorithm included the line "Simulate $P(x)$ for k steps". From our discussion of universal machines, we know that simulating another algorithm (including simulating another algorithm for a fixed number of steps) is something that algorithms can do. Therefore, we will allow this "simulation" operation as a single line of pseudocode, even if actually implementing it in the RAM model would be complicated. Similarly since we are not discussing the details of encodings that we are using in this chapter, we will allow algorithms to convert between objects and their encodings without specifying how this is done. For example, if we have an algorithm that takes as input $\langle P\rangle$, the encoding of an algorithm P, we could write a line like

$$n \leftarrow \text{ the number of instructions in } P$$

without discussing how we obtained the number of instructions from the string $\langle P\rangle$.

It's important to understand that whether or not a language is decidable does not depend on whether we *know* an algorithm which decides the language, and in fact it is possible to prove that a language is decidable without constructing a specific algorithm which decides it, as the following problem demonstrates (we call such a proof "non-constructive").

Theorem 31. *Let*

$$L = \{\langle n\rangle \mid n \in \mathbb{Z}_{\geq 0} \textit{ and the base-10 expansion of } \pi \textit{ contains } n \textit{ consecutive } 7s\}$$

Then L is decidable.

Proof. We consider two cases, and show that in either case, an algorithm that decides this language exists.

- First, suppose that for every positive integer k, there is a sequence of k consecutive 7s somewhere in the decimal expansion of π. In this case, L is decided by the following algorithm:

```
L-DECIDER(n)
    Return True
```

- Now suppose that there is some integer k such that the decimal expansion of π does not contain a sequence of k consecutive 7s. Choose the minimum such k. By our choice of k, π contains a sequence of $k-1$ consecutive 7s, and this sequence contains within it a sequence of i consecutive 7s for each $i < k$ (simply take the first i elements of the sequence). On the other hand, π cannot contain a sequence of i consecutive 7s for any $i \geq k$, because this sequence would contain a sequence of k consecutive 7s, and by our choice of k, no such sequence exists. Therefore, L is decided by the following algorithm:

```
L-DECIDER(n)
    If n < k :
        Return True
    Return False
```

Note that we do not know which of these cases holds (and in the second case we don't know the value of k), and thus we don't know which of these algorithms decides L. Nevertheless, we know that one of the algorithms we described above must decide L, so L is decidable even if we aren't sure which of the algorithms we presented is the one that actually decides it.

Sidenote: Even though π is transcendental, it's actually an open problem whether or not its decimal expansion contains strings of 7s (or any other digit) of arbitrary length. □

6.3 The Halting Problem

Before Turing's famous 1936 paper it was widely believed that every problem could be solved by a sufficiently clever algorithm—i.e. that every language was decidable. One of Turing's largest contributions was proving that there exist undecidable languages—problems that algorithms cannot solve. Formally we say that a language is **undecidable** if it is not decidable. In this section we will see the first and most famous undecidable language and see how to prove that it is undecidable.

We define

$$\text{HALTS} = \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ halts}\}.$$

That is, HALTS is a language of algorithm-string pairs where the algorithm halts (outputs a value) when run on the string. This would be a useful language to have a decider for. For instance, when writing a homework autograder, it would be useful to have a way to distinguish between programs that loop forever versus ones that are very slow.

Intuitively, we might believe that HALTS should be decidable by some kind of brute force approach. Perhaps we can just run the algorithm for a long time and see if it ever ends up on the same instruction with the same data in its variables and in memory? Repeating all of these things would indeed imply that the algorithm is looping. Unfortunately since a RAM model program can expand its memory an arbitrary number of times, it's possible for a RAM model program to loop without ever repeating the same exact memory contents (A trivial example is a program that just repeatedly calls Malloc forever). There is no way we can brute force over the infinitely many potential memory configurations that may occur while running a given algorithm, so this approach to deciding HALTS does not work.

Stronger evidence that the Halting problem must be hard comes from the fact that an algorithm which decides HALTS could solve essentially every open question in mathematics. Proofs in mathematics can be written in an extremely formal language which is verifiable by a computer. For example, a number of important mathematics proofs have been written and verified in Lean. In principle, any mathematical proof could be written in Lean (although the proof might be very long), and we could construct an algorithm that takes a string and determines whether or not it is a Lean proof of a particular theorem. For example, we could create the following algorithm:

```
RIEMANN(x)
  For i from 0 to ∞
      For each string s ∈ Σ^i
          If s encodes a proof of the Riemann Hypothesis in Lean:
              Return True
```

The Riemann Hypothesis is one of the most famous unsolved problems in mathematics, with a one million dollar prize for whoever solves it. If the Riemann hypothesis is provable (e.g. because it is true), then it must have a Lean proof, which has some finite length n. Since there are only finitely many strings of length at most n, our algorithm RIEMANN will check all of them in finite time, find the proof of the Riemann hypothesis and return True. On the other hand if the Riemann Hypothesis is false, then there cannot be a proof of it, so RIEMANN will loop forever. Thus, if we could decide whether or not RIEMANN halts (the choice of input does not matter here, since RIEMANN ignores its input string x), then we could determine whether or not the Riemann hypothesis is true. There's nothing special about the Riemann hypothesis for this purpose either, we could construct a similar algorithm for any mathematical conjecture. The idea of a single program which can solve every mathematical conjecture may seem a bit too good to be true.

Indeed, Turing proved the following theorem:

Theorem 32. HALTS *is undecidable.*

Proof. We prove this by contradiction.

Suppose for contradiction that HALTS is decidable. Then, by the definition of a decidable language, there exists an algorithm HALTS-DECIDER which decides HALTS. The key concept we will use to obtain a contradiction is running an algorithm on its own encoding (source code). We construct a new algorithm CONTRARY which takes as input an algorithm P, then asks HALTS-DECIDER whether the algorithm halts when run on its own source code. If HALTS-DECIDER says that $P(\langle P \rangle)$ halts, then CONTRARY enters an infinite loop. If HALTS-DECIDER says that $P(\langle P \rangle)$ loops forever, then CONTRARY immediately halts.

```
CONTRARY(P)
  If HALTS-DECIDER(P, ⟨P⟩) returns True:
      Loop forever
  Return True
```

Now consider what happens if we run CONTRARY($\langle$CONTRARY$\rangle$). Then CONTRARY will call HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$). We consider two cases:

- Suppose that $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \in$ HALTS. Then, since HALTS-DECIDER is a decider for HALTS, HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$) returns True. Thus, CONTRARY($\langle$CONTRARY$\rangle$) will reach the "Loop forever" line and loop forever. Therefore $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \notin$ HALTS. This contradicts what we assumed at the start of this case, so this case cannot occur.
- Now suppose that $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \notin$ HALTS. Then, since HALTS-DECIDER is a decider for HALTS, HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$) returns False. Therefore, CONTRARY($\langle$CONTRARY$\rangle$) will reach the "Return True" line and halt.
  Thus, $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \in$ HALTS. This contradicts what we assumed at the start of this case, so this case cannot occur either.

Thus, whether $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle$ is in HALTS or not, we get a contradiction, meaning that our initial assumption that HALTS-DECIDER exists must have been false, and HALTS is undecidable. □

Note that HALTS is semi-decidable, as shown by the following simple algorithm that accepts HALTS:

```
HALTS-ACCEPTOR(P, x)
  Run P(x)
  Return True
```

The "Run $P(x)$" line will eventually finish running if and only if $P(x)$ halts. Thus, if $P(x)$ halts, HALTS-ACCEPTOR returns True, and if $P(x)$ loops forever then so does HALTS-ACCEPTOR.

6.4 Comprehension Questions

1. True or false? Every acceptable language is decidable.
2. Suppose that U is a universal machine. We define two languages:

$$L_1 := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns True}\}$$
$$L_2 := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ halts}\}$$

   Which of the following is true?

   (a) U decides the language L_1.

   (b) U decides the language L_2.

   (c) U accepts the language L_1, but does not decide it.

(d) U accepts the language L_2, but does not decide it.

(e) U does not accept or decide any of these languages.

3. True or false? The language $L := \{\langle n \rangle : n \in \mathbb{Z}_{\geq 0}$ and the base-10 expansion of π contains at most n consecutive 0s$\}$ is decidable.

4. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ (i.e. the digits 0-9 as well as a decimal point). Which of the following sets are languages? (Select all that apply)

   (a) Σ^*

   (b) $\mathbb{R}^+$

   (c) $\mathbb{Z}^+$

   (d) $\{\langle x \rangle \mid x \in \mathbb{Z}^+\}$

5. True or false? The language HALTS is acceptable.

6. Two of the following four languages are decidable. Determine which two.

   (a)
   $$\text{HAS-MULT} := \{\langle P \rangle \mid P \text{ is an algorithm which includes an instruction of the form } i \leftarrow j * k\}$$

   (b)
   $$\text{USES-MULT} := \{\langle P, x \rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ executes an instruction of the form } i \leftarrow j * k\}$$

   (c) NOT-HALTS $:= \Sigma^* \setminus$ HALTS

   (d) HALTS-FAST $:= \{\langle P, x \rangle \mid P$ is an algorithm, $x \in \Sigma^*$ and $P(x)$ halts after at most 100 steps$\}$

6.5 Exercises

1. Prove that if L_1 and L_2 are two decidable languages then $L_1 \cup L_2$ is decidable.

2. Prove that there exist two undecidable languages L_1 and L_2 such that $L_1 \cup L_2$ is decidable.

3. Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. Prove that if L and $\Sigma^* - L$ are both semi-decidable, then L is decidable.

4. Prove that the following language is decidable:

   NO-MAL-HALTS $:= \{\langle P, x \rangle \mid P$ is an algorithm, $x \in \Sigma^*$ and $P(x)$ halts without calling Malloc.$\}$

5. A graviton is a hypothetical elementary particle. Physicists are currently not sure whether or not gravitons exist. Prove that the following language is decidable (no knowledge of physics required):
   $$\text{GRAVITONS} := \begin{cases} \{\text{“yes”}\} & \text{if gravitons exist} \\ \{\text{“no”}\} & \text{otherwise} \end{cases}$$

6. We say that a RAM model program P is “pretty small” if it satisfies the following conditions:

   - P's alphabet is $\Sigma = \{0, 1\}$
   - P has at most 411 instructions.
   - All integer constants used in P are between 0 and 411.

   Prove that the following language is decidable:
   $$\text{SMALL-HALTS} := \{\langle P \rangle \mid P \text{ is a “pretty small” program and } P(\varepsilon) \text{ halts}\}$$
   (Hint: How many meaningfully distinct “pretty small” programs are there)

7 Undecidability Reductions

In the previous chapter we saw our first example of an undecidable language: HALTS. In this chapter we will discuss *reductions*, a powerful technique for relating a new problem to one that we've already solved which we will use throughout the rest of the course. In Section 7.1 we will define reductions and use them to show that a variety of languages about the encodings of algorithms are undecidable. In Section 7.2, we will introduce an undecidable problem which is *not* about the encodings of algorithms called the Post Correspondence Problem, and show how PCP can also be used as a basis for undecidability reductions. Finally in Section 7.3 we will extend our discussion of the limits of computation beyond algorithms that return a boolean by defining uncomputable functions and showing that a particular function on the integers is uncomputable.

7.1 Turing Reductions

One of the most powerful concepts in computer science is *reduction*. Reducing problem A to problem B means giving an algorithm to solve A, assuming that you already have an algorithm for B. Reductions are used in everyday life. For example, if I am lost in Texas I can reduce the problem of getting home to the problem of finding Highway 6 (since I already know how to navigate to my apartment from Highway 6). They are also commonly used in programming. For example, say that you write a function to find the minimum element of an array by calling a "sort" function from the standard library and then taking the first element of the array. You have reduced the problem of finding the minimum element to the problem of sorting an array. Reductions are more than just a technique for solving new problems, however. Reductions give us a way to compare the "difficulty" of problems, and as we'll see in this section, they are a useful tool for proving that a language is undecidable.

For two languages A and B we say that $A \leq_T B$ (read "A **Turing reduces to** B") if it is possible to construct a decider for A using a decider for B as a subroutine. We use a less than or equal symbol because Turing reductions allow us to compare the difficulty with respect to decidability. In particular, if $A \leq_T B$ then we can say the following:

1. If B is decidable, then so is A.
2. If A is undecidable, then so is B.

Conclusion 1 follows from the definition of a Turing reduction. If $A \leq_T B$, then we can construct a decider for A given a decider for B, and if B is decidable then we have a decider for B. Conclusion 2 is simply the contrapositive of conclusion 1.

Theorem 33. *The language*

$$\text{RET-TRUE} := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns True}\}$$

is undecidable.

Proof. We could use the same technique as in Theorem 32 to prove this language undecidable, but in the interest of demonstrating reductions, we will instead prove this by showing that

$$\text{HALTS} \leq_T \text{RET-TRUE}.$$

We need to construct a decider for HALTS, assuming that we have access to a decider for RET-TRUE. Let RTDecider be a decider for RET-TRUE. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate P(y)
            Return True
    Return RTDecider(Helper, x)
```

To show that HaltsDecider correctly decides HALTS we need to show that for every $\langle P, x\rangle \in$ HALTS, HaltsDecider(P, x) returns True and for every $\langle P, x\rangle \notin$ HALTS, HaltsDecider(P, x) returns False.

- Suppose that $\langle P, x\rangle \in$ HALTS. Then $P(x)$ halts, so Helper(x)'s simulation of $P(x)$ will finish and Helper(x) will return True. Therefore $\langle$Helper$, x\rangle \in$ RET-TRUE and thus RTDecider(Helper$, x)$ returns True, so HaltsDecider(P, x) also returns True.
- Now suppose that $\langle P, x\rangle \notin$ HALTS. Then $P(x)$ loops, so Helper(x)'s simulation of $P(x)$ will loop forever and Helper(x) will never return True. Therefore $\langle$Helper$, x\rangle \notin$ RET-TRUE and thus RTDecider(Helper$, x)$ returns False, so HaltsDecider(P, x) also returns False.

Thus, we have shown that HALTS $\leq_T$ RET-TRUE, so since HALTS is undecidable, RET-TRUE is also undecidable. □

This proof may seem a little counterintuitive. After all, we constructed a decider for HALTS which we just showed in the previous chapter is impossible! This reduction is essentially just another form of proof by contradiction. We showed that if RET-TRUE were decidable, then HALTS would be as well. Since we know HALTS is not decidable, neither is RET-TRUE.

It's also important to recognize that our decider for HaltsDecider constructed a helper algorithm Helper, *but never actually ran* Helper. If HaltsDecider simulated running Helper (or P) we couldn't say that HaltsDecider is a decider, because the simulation of P might cause HaltsDecider to loop forever. But since Helper is never run, just written as source code that gets passed into RTDecider (which always halts because we assumed it is a decider), HaltsDecider is guaranteed to always return an answer without infinite looping.

Theorem 34. *The language*

$$\text{EMPTY} := \{\langle P\rangle \mid P \text{ is an algorithm and } L(P) = \emptyset\}$$

is undecidable.

Proof. We prove this by showing that

$$\text{HALTS} \leq_T \text{EMPTY}.$$

Let EmptyDecider be a decider for EMPTY. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate running P(x)
            Return True
    Return ¬EmptyDecider(Helper)
```

Now we prove that HaltsDecider decides HALTS.

- Suppose that $\langle P, x\rangle \in$ HALTS. Then for every $y \in \Sigma^*$, Helper(y) finishes simulating $P(x)$ and therefore returns True. Thus $L($Helper$) = \Sigma^* \neq \emptyset$, so $\langle$Helper$\rangle \notin$ EMPTY, so EmptyDecider(Helper) returns False, so HaltsDecider(P, x) returns True.
- Suppose that $\langle P, x\rangle \notin$ HALTS. Then for every $y \in \Sigma^*$, Helper(y) never finishes simulating $P(x)$ and therefore loops forever. Thus $L($Helper$) = \emptyset$, so $\langle$Helper$\rangle \in$ EMPTY, so EmptyDecider(Helper) returns True, so HaltsDecider(P, x) returns False.

□

Notice that once again, our machine HaltsDecider created a helper machine Helper without actually running it. This time Helper ignores its own input string and runs $P(x)$ (rather than $P(y)$) so that the behavior of Helper on *all* strings depends on the behavior of M on the specific string x. The other major difference compared to the previous reduction is that HaltsDecider is doing the *opposite* of what EmptyDecider does, because $P(x)$ halting corresponds to $\langle \text{Helper} \rangle \notin \text{EMPTY}$.

Theorem 35. *The language*

$$\text{ACCEPTS-2} := \{\langle P \rangle \mid P \text{ is an algorithm and } |L(P)| = 2\}$$

is undecidable.

Proof. We prove this by showing that

$$\text{HALTS} \leq_T \text{ACCEPTS-2}.$$

Let A2Decider be a decider for ACCEPTS-2. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate running P(x)
            If y = ε or y = "411":
                Return True
            Return False
    Return A2Decider(Helper)
```

Now we prove that HaltsDecider decides HALTS.

- Suppose that $\langle P, x \rangle \in \text{HALTS}$. Then for every $y \in \Sigma^*$, Helper(y) finishes simulating $P(x)$. Therefore Helper(y) returns True ε and "411" and returns False on all other inputs. Thus $L(\text{Helper}) = \{\varepsilon, \text{"411"}\}$, so $|L(\text{Helper})| = 2$, so $\langle \text{Helper} \rangle \in \text{ACCEPTS-2}$, so A2Decider(Helper) returns True, so HaltsDecider(P, x) returns True.
- Suppose that $\langle P, x \rangle \notin \text{HALTS}$. Then for every $y \in \Sigma^*$, Helper(y) never finishes simulating $P(x)$ and therefore loops forever. Thus $L(\text{Helper}) = \emptyset$, so $\langle \text{Helper} \rangle \notin \text{ACCEPTS-2}$, so A2Decider(Helper) returns False, so HaltsDecider(P, x) returns False.

□

At this point you may be seeing the general pattern. To prove that some language L containing the encodings of algorithms is undecidable, we show that $\text{HALTS} \leq_T L$. To prove this we construct a decider for HALTS. This decider for HALTS constructs a helper machine which runs $P(x)$ and then returns True if its input y is in some set $S \subseteq \Sigma^*$ (where the choice of S depends on L). If $\langle P, x \rangle \in \text{HALTS}$, then $L(\text{Helper}) = S$, otherwise $L(\text{Helper}) = \emptyset$. If machines with language S are in L and machines with language $\emptyset$ are not, or vice-versa, then we can decide HALTS by running our decider for L on Helper and returning either the same thing (if $\langle \text{Helper} \rangle$ is in L when $L(\text{Helper}) = S$) or the opposite (if $\langle \text{Helper} \rangle$ is in L when $L(\text{Helper}) = \emptyset$).

This general pattern can work even for some languages that look more complicated, like the following one which involves pairs of algorithms:

Theorem 36. *The language*

$$\text{UNIQUE-INTERSECT} := \{\langle P_1, P_2 \rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } |L(P_1) \cap L(P_2)| = 1\}$$

is undecidable.

Proof. To prove that UNIQUE-INTERSECT is undecidable, we show that

$$\text{HALTS} \leq_T \text{UNIQUE-INTERSECT}.$$

Let UIDecider be a decider for UNIQUE-INTERSECT. We create the following decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Run P(x)
            If y = ε:
                Return True
            Else:
                Return False
    Return UIDecider(Helper, Helper)
```

Correctness:

- Suppose that $\langle P, x\rangle \in$ HALTS. Then $L(\text{Helper}) = \{\varepsilon\}$, so $|L(\text{Helper}) \cap L(\text{Helper})| = |\{\varepsilon\}| = 1$. Thus $\langle \text{Helper}, \text{Helper}\rangle \in$ UNIQUE-INTERSECT, so UIDecider(Helper, Helper) will return True, so HaltsDecider(P, x) will return True.
- Suppose that $\langle P, x\rangle \notin$ HALTS. Then $L(\text{Helper}) = \emptyset$, so $|L(\text{Helper}) \cap L(\text{Helper})| = |\emptyset| = 0$. Thus $\langle \text{Helper}, \text{Helper}\rangle \notin$ UNIQUE-INTERSECT, so UIDecider(Helper, Helper) will return False, so HaltsDecider(P, x) will return False.

□

So far all of the reductions we have seen have been from HALTS. This isn't the only language we can reduce from, however. To prove that a language is undecidable, we can reduce from any language that already know is undecidable (so far we've added RET-TRUE, EMPTY, ACCEPTS-2, and UNIQUE-INTERSECT to that list).

Theorem 37. *The language*

$$\text{FINITE} := \{\langle P\rangle \mid P \text{ is an algorithm and } L(P) \text{ is finite}\}$$

is undecidable.

Proof. To prove that FINITE is undecidable, we will show that

$$\text{EMPTY} \leq_T \text{FINITE}.$$

Let FiniteDecider be a decider for FINITE. We create the following decider EmptyDecider for EMPTY:

```
EmptyDecider(P)
    Σ ← the alphabet of P
    Write out the source code of the following algorithm Helper:
        Helper(x)
            For num_steps from 1 to ∞:
                For n from 0 to num_steps:
                    For each s ∈ Σ^n:
                        Simulate P(s) for num_steps steps
                        If the simulation returned True:
                            Return True
    Return FiniteDecider(Helper)
```

Correctness:

- Suppose that $\langle P \rangle \in$ EMPTY. Then P never returns True for any input, so no matter what string s HELPER simulates P on, $P(s)$ will never return True, and therefore HELPER(x) will never return True for any x. Thus $L(\text{HELPER}) = \emptyset$. Therefore $|L(\text{HELPER})| = |\emptyset| = 0$, which is clearly finite, so $\langle \text{HELPER} \rangle \in$ FINITE. Therefore FINITEDECIDER(HELPER) will return True, so EMPTYDECIDER(P) will return True.
- Suppose that $\langle P \rangle \notin$ EMPTY. Then there exists some $y \in \Sigma^*$ such that $P(y)$ returns True. Let t be the number of steps that it takes for $P(y)$ to return True. We note that each iteration of the outer for loop of HELPER eventually finishes (i.e. runs in finite time) because there are only a finite number of strings of length at most num_steps, and we run P for only a finite number of steps on each of them. Thus, if it has not already returned True, HELPER will eventually reach an iteration of the outer for loop where num_steps is $\max(|y|, t)$. In this iteration, it will simulate $P(y)$ for at least t steps, and therefore the simulation will return True and so HELPER will return True.

  Therefore, HELPER will return True for every input string (since HELPER does the same thing for every input string x), and thus $L(\text{HELPER}) = \Sigma^*$, which is infinite. Therefore $\langle \text{HELPER} \rangle \notin$ FINITE, so FINITEDECIDER(HELPER) will return False, and therefore EMPTYDECIDER(P) will return False.

□

In this case, reducing from EMPTY did not make it easier to show that FINITE was undecidable (see if you can find an alternative proof using a reduction from HALTS). In the next section, we'll see an example of a problem where HALTS is actually not the best problem to reduce from to prove undecidability.

However, this proof did allow us to show off a technique for running a program on infinitely many strings in parallel (despite the fact that our model of computation is inherently sequential!). The approach that we used in this proof's helper machine of running P on increasingly large sets of strings for increasingly large numbers of steps (which guarantees that if there is any string for which P returns True, we will find it in some finite amount of time) is called **dovetailing**, and it occasionally comes up in other problems in complexity theory.

Note that our reduction would not have worked if we had tried to use a simpler helper machine like the one below which runs P on strings one at a time:

```
HELPER(x)
  For n from 0 to ∞:
      For each s ∈ Σ^n:
          Simulate P(s)
          If the simulation returned True:
              Return True
```

If, for example, $P(\varepsilon)$ loops forever, then this version of the helper algorithm will get stuck simulating $P(\varepsilon)$ forever, and never move on to longer strings. Thus, even if P returns True on some other string, this version of the helper machine would never find it. That's why we needed the dovetailing technique.

7.2 The Post Correspondence Problem

Not all undecidable languages are about encodings of algorithms. In 1946, Emil Post described the following problem, which became known as the Post Correspondence problem, and showed that it cannot be solved by any algorithm.

In the **Post correspondence problem**, we are given some finite set of dominoes where each domino has a string written on the top and the bottom. For example, we might be given the set:

$$\left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\}.$$

The goal of the problem is to find a sequence of one or more copies of dominoes from this set such that the concatenation of the strings on the top of the dominoes is the same as the concatenation of the strings on the bottom of the dominoes. We call such a sequence a **match**. For example, the following arrangement would be a match for the set of dominoes above:

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right].$$

Both the top and the bottom strings are *abcaaabc*. On the other hand the set of dominoes below has no match:

$$\left\{\left[\frac{ab}{a}\right], \left[\frac{cab}{bc}\right], \left[\frac{a}{\varepsilon}\right], \left[\frac{aaa}{b}\right], \left[\frac{dcb}{bca}\right]\right\}.$$

To see this we note that we cannot use the fifth domino, because it has a d on top and none of the dominoes have a d on the bottom, and we cannot form a solution out of the first four dominoes, because on each of them the top string is longer than the bottom string, so we cannot have a sequence of dominoes where the top and bottom strings are even the same length, let alone equal to each other.

Theorem 38. *The language*

$$\text{PCP} := \{\langle P\rangle \mid P \text{ is an instance of the Post correspondence problem with a solution}\}$$

is undecidable.

We will not prove this theorem, because a full proof would take several pages and get deep into the nitty-gritty instruction sets of RAM model programs. The very high level idea is to prove that HALTS $\leq_T$ PCP by constructing a set of dominoes for which any solution to the problem would have the matching top and bottom string represent the sequence of instructions and configurations that P goes through when run on x. This sequence of configurations only terminates (and thus corresponds to a finite sequence of dominoes) if $P(x)$ eventually halts.

Even though we did not prove it ourselves, we can use Theorem 38 to prove the undecidability of other languages.

Theorem 39. *The language*

$$\begin{aligned}\text{MPCP} := \{\langle P, d\rangle \mid\ & P \text{ is an instance of the Post correspondence problem,}\\ & d \text{ is a domino in } P \text{ and } P \text{ has a solution beginning with } d\}\end{aligned}$$

is undecidable.

Proof. To prove that MPCP is undecidable, we will show that PCP $\leq_T$ MPCP. Let MPCPDecider be a decider for MPCP. We create the following decider PCPDecider for PCP:

```
PCPDecider(P)
    For each domino d in P:
        If MPCPDecider(P, d) returns True:
            Return True
    Return False
```

□

Correctness:

- Suppose $\langle P\rangle \in$ PCP. Then P has a match, and this match must begin with some domino x from P. Since there are only finitely many dominoes in P and MPCPDecider(P, d) runs in finite time for each of them (since MPCPDecider is a decider), the for loop in PCPDecider will eventually reach an iteration where $d = x$ and it will run MPCPDecider(P, x). By our choice of x, MPCPDecider(P, x) returns True, so PCPDecider(P) will return True.

- Suppose $\langle P \rangle \notin$ PCP. Then P has no solution, so it does not have a solution starting with d for any domino d in P. Therefore all of the calls that PCPDecider makes to MPCPDecider will return False, so PCPDecider(P) will return False.

PCP has been used as a starting point to prove several other more interesting languages undecidable, although the proofs are beyond the scope of this course.

7.3 Uncomputable Functions

So far in this chapter and the previous one, we have focused on programs that only return True or False (or loop forever) in order to prove that languages are decidable or undecidable. The idea of decidability can naturally be extended to arbitrary functions, however. We say that a function $f : X \to Y$ is **computable** if there is some algorithm that computes it, i.e. if there exists an algorithm P such that for every $x \in X$, $P(\langle x \rangle)$ outputs $\langle f(x) \rangle$ (We do not care what P does when its input string does not encode an element of f's domain X. We can generally just assume that the algorithm returns ε in this case). We say that a function is **uncomputable** if it is not computable.

While Turing reductions are only defined between languages, we can prove that a function is uncomputable using a similar technique: assume for sake of contradiction that the function is computable, then use the algorithm which computes the function to create a decider for HALTS (or some other known undecidable language). Since HALTS is undecidable, this is a contradiction, and so our assumption that the function was computable must have been false.

Let's see an example of this technique. We define a **binary RAM model program** as a RAM model program with alphabet $\Sigma = \{0, 1\}$ which does not use any integer constants other than 0 and 1. We define the **busy beaver function**, $BB : \mathbb{Z}^+ \to \mathbb{Z}^+$ as follows: $BB(n)$ is the maximum number of steps that an n-instruction binary RAM model program can take on input ε before halting (here the max is across all n-instruction binary RAM model programs which halt on input ε).

Theorem 40. *BB is uncomputable.*

Proof. AFSOC that BB is computable. Let Beaver be an algorithm that computes BB.

We construct the following decider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of a binary RAM model program Helper:
        Helper(y)
            Return P(x)
    n ← the number of instructions in Helper
    t ← Beaver(n)
    Simulate Helper(ε) for t steps
    If the simulation of Helper halted:
        Return True
    Return False
```

We note that we have assumed here that we can simulate P using a helper algorithm which is a *binary* RAM model program. i.e. we have assumed that there exists a universal machine which uses $\Sigma = \{0, 1\}$ and uses no integer constants besides $\{0, 1\}$. This is true, and you may assume this fact in your proofs, but we will not prove it, since we did not even formally prove that universal machines exist in the first place.

We will now prove that HaltsDecider correctly decides HALTS:

- Suppose that $\langle P, x \rangle \in$ HALTS. Then Helper(ε) will halt. Since Helper is a binary RAM model program with n instructions, we know that it halts in at most $BB(n)$ steps. Since Beaver(n) computes $BB(n)$, this means that the simulation of Helper(ε) will halt within the t steps that we run it for, so HaltsDecider(P, x) will return True.

- Suppose that $\langle P, x \rangle \notin$ HALTS. Then Helper(ε) loops forever, so when we simulate it for t steps it will not halt. Thus HaltsDecider(P, x) will return False.

□

The busy beaver function is traditionally defined in terms of Turing machines. Since we have avoided describing the technical details of Turing machines, in this course we will use our version based on binary RAM model programs. Note that the restriction to programs that only use the constants 0 and 1 is necessary for the function to be well-defined, because if we allow arbitrarily large constants then we can make small programs that run arbitrarily long. For example, in the following program we can set c to any positive constant to obtain a 5-line program that runs for more than $3c$ steps.

1: i $\leftarrow$ 0
2: Malloc('0')
3: i $\leftarrow$ i + 1
4: If i < c: goto 2
5: Output(1,1)

We can create a binary RAM model program with the same functionality as the one above, but only by letting the number of instructions grow in proportion to c (i.e. by replacing the loop with c copies of lines 2 and 3).

7.4 Comprehension Questions

1. Suppose that A and B are languages and $A \leq_T B$. Which of the following can we conclude? (select all that apply)

   (a) If A is decidable, then B is decidable.

   (b) If B is decidable, then A is decidable.

   (c) If A is undecidable, then B is undecidable.

   (d) If B is undecidable, then A is undecidable.

2. Let IS-EVEN $:= \{\langle n \rangle \mid n \text{ is an even integer}\}$. Let SUBSET-SUM $:= \{\langle S, k \rangle \mid S \subseteq \mathbb{Z}, k \in \mathbb{Z}, \exists T \subseteq S, \sum_{x \in T} x = k\}$. Which of the following are true?

   (a) IS-EVEN $\leq_T$ HALTS

   (b) IS-EVEN $\leq_T$ SUBSET-SUM

   (c) SUBSET-SUM $\leq_T$ HALTS

   (d) SUBSET-SUM $\leq_T$ IS-EVEN

   (e) HALTS $\leq_T$ IS-EVEN

   (f) HALTS $\leq_T$ SUBSET-SUM

3. Define a language RET-FALSE $:= \{\langle P, x \rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns False}\}$. Suppose we wish to show that HALTS $\leq_T$ RET-FALSE. We assume that we have a decider RFDecider for RET-FALSE and give the following decider for HALTS:

   ```
   HaltsDecider(P, x)
     Write down the source code of an algorithm Helper.
     Return RFDecider(Helper, x)
   ```

   To finish this proof we need to describe how to design Helper. Which of the following choices for Helper would allow HaltsDecider to correctly decide HALTS?

   (a)
   ```
   Helper(y)
     Run P(x)
     Return True
   ```

(b)
```
HELPER(y)
  Run P(x)
  Return False
```

(c)
```
HELPER(y)
  If y = x:
      Return False
  Run P(x)
  Return True
```

(d)
```
HELPER(y)
  If y = x:
      Return True
  Run P(x)
  Return False
```

(e)
```
HELPER(y)
  Run P(x)
  Return y = x
```

4. Suppose that we have an instance of the Post correspondence problem with the following 5 dominoes:

   (a) $\left[\frac{a}{bc}\right]$
   (b) $\left[\frac{b}{ab}\right]$
   (c) $\left[\frac{ca}{a}\right]$
   (d) $\left[\frac{cba}{cb}\right]$
   (e) $\left[\frac{acb}{ab}\right]$

   Which of these dominoes must come first in any match for this instance of the problem?

5. Let P be an instance of PCP and let d be a domino from P. Which of the following statements are guaranteed to be true? (select all that apply)

   (a) If $\langle P \rangle \in$ PCP then $\langle P, d \rangle \in$ MPCP.
   (b) If $\langle P, d \rangle \in$ MPCP then $\langle P \rangle \in$ PCP.

6. True or false? $\forall\, n \in \mathbb{Z}^+, \exists\, t \in \mathbb{Z}^+$, every n instruction binary RAM model program which halts on input ε halts on ε after at most t steps.

7.5 Exercises

1. Prove that the following language is undecidable:

$$L := \{\langle P \rangle \mid P \text{ is an algorithm and there exist at least two distinct input strings } x, y \in \Sigma^* \text{ such that } P(x) \text{ and } P(y) \text{ loop forever.}\}$$

2. Define

   REJECTS-3 := $\{\langle P \rangle \mid P$ is an algorithm which returns False for exactly 3 different input strings$\}$

   Prove that REJECTS-3 is undecidable.

3. Define

$$\text{EQ} := \{\langle P_1, P_2 \rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } L(P_1) = L(P_2)\}.$$

   Prove that EQ is undecidable.

4. Define

$$\text{DISJOINT} := \{\langle P_1, P_2\rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } L(P_1) \cap L(P_2) = \emptyset\}.$$

Prove that DISJOINT is undecidable.

5. Prove that RET-TRUE $\leq_T$ HALTS
6. Prove that EMPTY $\leq_T$ HALTS
7. Prove that the following language is undecidable

$$\begin{aligned}\text{ODD-PCP} :=\{\langle P\rangle \mid\ & P \text{ is an instance of the Post correspondence problem,}\\ & \text{which has a solution using an odd number of total dominoes}\}\end{aligned}$$

8. Prove that the following language is undecidable:

$$\begin{aligned}L :=\{\langle P\rangle \mid\ & P \text{ is an instance of the Post correspondence problem where}\\ & \text{every non-empty top/bottom string on a domino begins}\\ & \text{with the symbol 'a' and } P \text{ has a match}\}\end{aligned}$$

9. Show that the function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ defined by
$f(n) :=$ the number of n-instruction binary RAM model programs which halt on ε
is uncomputable.
10. Show that the function $g : \{P \mid P \text{ is an algorithm}\} \times \mathbb{Z}^+ \to \mathbb{Z}^+$ defined by
$g(P, n) :=$ the number of strings $x \in \Sigma^n$ for which $P(x)$ returns True
is uncomputable.
11. Show that the function $h : \{P \mid P \text{ is an binary RAM model program}\} \to \mathbb{Z}^+$ defined by
$h(P) :=$ The number of instructions in the shortest binary RAM model program with the same language as P (i.e. the P' with the fewest instructions for which $L(P') = L(P)$)
is uncomputable.
12. We define the following languages:

$$\begin{aligned}\text{MM3} :=\{\langle S\rangle : & S \text{ is a set of } 3\times 3 \text{ matrices with integer entries such that it is possible}\\ & \text{to obtain the zero matrix by multiplying a finite sequence of matrices from } S\}\\ \text{MM4} :=\{\langle S\rangle : & S \text{ is a set of } 4\times 4 \text{ matrices with integer entries such that it is possible}\\ & \text{to obtain the zero matrix by multiplying a finite sequence of matrices from } S\}\end{aligned}$$

For example, suppose S is a set containing the following matrices:

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}, \begin{bmatrix}-1&-2&-3\\4&5&6\\-7&-8&-9\end{bmatrix}, \begin{bmatrix}1&1&0\\0&0&1\\0&0&1\end{bmatrix}, \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}$$

Then $\langle S\rangle \in$ MM3 because

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&1&0\\0&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix} = \begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$$

MM3 is known to be undecidable (you may assume this). Prove that MM4 is undecidable.