CSCI 4602
Fall 2020
Lecture Notes

Contents

Module 4: Polynomial-Time Computability

Practice Questions for All Sections

1 Review of Propositional Logic

This section reviews propositional logic, which you should already have seen.

1.1 Syntax of propositional logic

The *syntax* of propositional logic only says what a propositional formula looks like. It does not say what a propositional formula means. We use A, B, C and ϕ (Greek phi) to name arbitrary propositional formulas.

Definition 1.1. A *propositional formula* is defined as follows.

1. Symbols **T** and **F** are propositional formulas.
2. A *propositional variable* is a propositional formula. We will use P, Q, R and S, possibly with subscripts, as propositional variables and X to refer to an arbitrary variable.
3. If A and B are propositional formulas then so are
   (a) $A \vee B$,
   (b) $A \wedge B$,
   (c) $\neg A$,
   (d) (A).

For example, each of the following is a propositional formula.

- P
- $P \vee Q$
- $P \wedge \neg Q$
- $P \wedge Q \wedge R$
- $Q \vee P \wedge R$
- $(R \wedge \mathbf{T}) \vee \neg Q$

Operator $\vee$ is read "or", $\wedge$ is read "and", and $\neg$ is read "not".

Rules of *precedence* and *associativity* determine how you break a propositional formula into subformulas. Higher precedence operators are done first. The following lists operators by precedence, from highest to lowest.

Precedence	
parentheses	high
$\neg$	
$\wedge$	
$\vee$	low

For example, $P\vee Q\wedge R$ is understood to have the same structure as $P\vee(Q\wedge R)$ since $\wedge$ has higher precedence than $\vee$.

Associativity determines how an expression is broken into subexpressions when it involves two or more occurrences of the same operator. We assume that operators $\vee$ and $\wedge$ are done from left to right. That is, they are *left-associative*. (Associativity is like the wind. A north wind blows from north to south.) For example, $P \vee Q \vee R$ has the same structure as $(P \vee Q) \vee R$. Associativity does not really matter for $\vee$ and $\wedge$ because they are *associative operators*. But associativity does matter for some operators, so it is wise to think about it.

1.2 Meaning of propositional logic

The meaning of a propositional formula can only be defined when the values of all of its variables are given. Each variable can be true or false.

Definition 1.2. A *truth-value assignment* is a set of components of the form $X = V$ where X is a variable and V is either T or F. For example, $\{P{=}\mathrm{T},\ Q{=}\mathrm{F}\}$ is a truth-value assignment. (Note that T and F are possible values of a propositional variable or a propositional formula. Do not confuse them with **T** and **F**, which are propositional formulas.)

Definition 1.3. If a is a truth-value assignment and X is a variable then $a(X)$ is the value (T or F) that a gives for variable X. For example, if a is $\{P$=T, Q=F$\}$ then $a(P)$ = T and $a(Q)$ = F.

Definition 1.4. Suppose that ϕ is a propositional formula and a is a truth-value assignment that defines every variable that occurs in ϕ. Notation $(a \dashv \phi)$ indicates the value of ϕ (either T or F) when variables have values given by a. Specifically:

1. $(a \dashv \mathbf{T})$ = T. That is, symbol **T** is always true.
2. $(a \dashv \mathbf{F})$ = F. That is, symbol **F** is always false.
3. If X is a variable then $(a \dashv X) = a(X)$. That is, X has the value that it is given by truth-value assignment a.
4. $(a \dashv A \vee B)$ is T if at least one of $(a \dashv A)$ and $(a \dashv B)$ is T, and is F otherwise. For example, $(\{P$=T, Q=F$\} \dashv P \vee Q)$ is T because $(\{P$=T, Q=F$\} \dashv P)$ is T, and we only need one of P and Q to be true.
5. $(a \dashv A \wedge B)$ is T if both of $(a \dashv A)$ and $(a \dashv B)$ are T, and is F otherwise. For example, $(\{P$=T, Q=F$\} \dashv P \wedge Q)$ is F because $(\{P$=T, Q=F$\} \dashv P)$ and $(\{P$=T, Q=F$\} \dashv Q)$ are not both T.
6. $(a \dashv \neg A)$ is T if $(a \dashv A)$ is F, and is F is $(a \dashv A)$ is T.
7. $(a \dashv (A)) = (a \dashv A)$. Parentheses only influence the structure of a propositional formula. A parenthesized formula (A) has the same meaning as A.

You determine the value of a propositional formula by building up larger and larger subexpressions, being careful to follow the rules of precedence and associativity. For example, suppose that $a = \{P$=F, Q=T, R=T$\}$. Then

(a) $(a \dashv Q)$ = T

(b) $(a \dashv P)$ = F

(c) $(a \dashv \neg P)$ = T by (b)

(d) $(a \dashv \neg P \wedge Q)$ = T by (a) and (c)

1.3 Additional definitions

Definition 1.5. $A \to B$ is defined to be an abbreviation for $\neg A \vee B$. Operator $\to$ is read "implies".

Intuitively, $A \to B$ means "if A is true then B is true." But that is not its definition. Its definition is that either A is false or B is true (or both). Notice that, if B is true, then $A \to B$ is true, *by definition*. Also, if A is false, then $A \to B$ is true, *by definition*.

Operator $\to$ has lower precedence than $\vee$ and is left-associative. Note that $\to$ is not an associative operator. $(A \to B) \to C$ does not have the same meaning as $A \to (B \to C)$.

Definition 1.6. $A \equiv B$ is defined to be the same as $(A \to B) \wedge (B \to A)$. Operator $\equiv$ is read "is equivalent to".

Formula $A \equiv B$ says that A and B have the same value; either both are true or both are false. In fact, $A \equiv B$ is equivalent to $(A \wedge B) \vee (\neg A \wedge \neg B)$. That is, either A and B are both true or A and B are both false.

Operator $\equiv$ has even lower precedence than $\to$. Here is a complete precedence table, from high to low precedence.

Precedence	
parentheses	high
$\neg$	
$\wedge$	
$\vee$	
$\to$	
$\equiv$	low

It is common to use an alternative name for $\equiv$.

Definition 1.7. $A \leftrightarrow B$ is the same as $A \equiv B$.

1.4 Truth tables

Since the value of a propositional formula depends on the values of its variables, one way to understand what the formula means is to look at its value for all possible values of the variables. That leads to the idea of a *truth table* of a propositional formula. The following is a truth table for $\neg P \vee Q$.

P	Q	$\neg$	P	$\vee$	Q
F	F	T	F	T	F
F	T	T	F	T	T
T	F	F	T	F	F
T	T	F	T	T	T

Under each variable, we write that variable's value. Under each operator, we write the value of the formula having that operator as its main or outermost operator. The column in blue is the value of the entire formula, $\neg P \vee Q$.

1.5 Validity

Definition 1.8. Propositional formula ϕ is *valid* if $(a \dashv \phi)$ is true for every truth value assignment a. A valid formula is also called a *tautology*.

For example, operator $\vee$ is commutative. Another way to say that is to say that formula

$$(P \vee Q) \equiv (Q \vee P)$$

is valid. Let's check that using a truth table.

P	Q	$(P$	$\vee$	$Q)$	$\equiv$	$(Q$	$\vee$	$P)$
F	F	F	F	F	T	F	F	F
F	T	F	T	T	T	T	T	F
T	F	T	F	F	T	F	T	T
T	T	T	T	T	T	T	T	T

The validity of

$$(P \vee Q) \equiv (Q \vee P)$$

is evident from the blue column of all Ts.

Table 1-1 shows a collection of propositional formulas that are all valid. You can check each one using a truth table.

Valid equivalences give you a way to replace one formula by another. For example, if you see $P \vee Q$ in any context, you can replace it by $Q \vee P$. In fact, you can replace any variable by any propositional formula in any of the above tautologies (or any other valid propositional formula) and they are still valid, provided (1) you replace every occurrence of a variable by the same propositional formula and (2) you use parentheses to avoid rules of precedence from rearranging the formula. For example, the commutative law for $\wedge$ says that

$$P \wedge Q \;\equiv\; Q \wedge P.$$

Replacing P by $(W \to V)$ and Q by $\neg R$ yields

$$(W \to V) \wedge \neg R \;\equiv\; \neg R \wedge (W \to V)$$

which is also valid. Review of propositional logic

Table 1-1: Some propositional tautologies	
Equivalence	**Name**
$\neg(\neg P) \;\equiv\; P$	double negation
$P \vee Q \;\equiv\; (Q \vee P)$	commutative law of $\vee$
$P \wedge Q \;\equiv\; (Q \vee P)$	commutative law of $\wedge$
$(P \vee Q) \vee R \;\equiv\; P \vee (Q \vee R)$	associative law of $\vee$
$(P \wedge Q) \wedge R \;\equiv\; P \wedge (Q \wedge R)$	associative law of $\wedge$
$(P \wedge (Q \vee R) \;\equiv\; (P \vee Q) \wedge (P \vee R)$	distributive law of $\wedge$ over $\vee$
$(P \vee (Q \wedge R) \;\equiv\; (P \wedge Q) \vee (P \wedge R)$	distributive law of $\vee$ over $\wedge$
$\neg(P \vee Q) \;\equiv\; \neg P \wedge \neg Q$	DeMorgan's law for $\vee$
$\neg(P \wedge Q) \;\equiv\; \neg P \vee \neg Q$	DeMorgan's law for $\wedge$
$\neg(P \rightarrow Q) \;\equiv\; P \wedge \neg Q$	DeMorgan's law for $\rightarrow$
$P \rightarrow Q \;\equiv\; \neg Q \rightarrow \neg P$	Law of the contrapositive
$(P \vee Q) \rightarrow R \;\equiv\; (P \rightarrow R) \wedge (Q \rightarrow R)$	cases
$(P \wedge Q) \rightarrow R \;\equiv\; (P \rightarrow (Q \rightarrow R))$	
$P \wedge \neg P \equiv \mathbf{F}$	contradiction 1
$P \;\equiv\; (\neg P \rightarrow P)$	contradiction 2
$P \;\equiv\; (\neg P \rightarrow \mathbf{F})$	contradiction 3
$P \vee \neg P$	Law of the excluded middle
$P \rightarrow P$	Law of the excluded middle, restated using $\rightarrow$
$\neg(P \wedge \neg P)$	Law of the excluded middle (DeMogan variant)
$P \rightarrow (Q \rightarrow P)$	
$\neg P \rightarrow (P \rightarrow Q)$	

2 Review of First-Order Logic

First-order logic (also called *predicate logic*) is an extension of propositional logic that is much more useful than propositional logic. It was created as a way of formalizing common mathematical reasoning. You should have seen first-order logic previously. This section is only review.

In first-order logic, you start with a nonempty set of values called the *universe of discourse* U. Logical statements talk about properties of values in U and relationships among those values.

2.1 Predicates

In place of propositional variables, first-order logic uses *predicates*.

Definition 2.1. A *predicate* P takes zero or more parameters x_1, x_2, …, x_n and yields either true or false. First-order formula $P(x_1, \ldots, x_n)$ is the value of predicate P with parameters x_1, …, x_n. A predicate with no parameters is a propositional variable. If P takes no parameters then P is a first-order formula.

Suppose that U is the set of all integers. Here are some examples of predicates. There is no standard collection of predicates that are always used. Rather, each of these is like a function definition in a computer program; different programs contain different functions.

- We might define $\text{even}(n)$ to be true if n is even. For example $\text{even}(4)$ is true and $\text{even}(5)$ is false.
- We might define $\text{greater}(x, y)$ to be true if $x > y$. For example, $\text{greater}(7, 3)$ is true and $\text{greater}(3, 7)$ is false.
- We might define $\text{increasing}(x, y, z)$ to be true if $x < y < z$. For example, $\text{increasing}(2, 4, 6)$ is true and $\text{increasing}(2, 4, 2)$ is false.

2.2 Terms

A *term* is an expression that stands for a particular value in U. The simplest kind of term is a *variable*, which can stand for any value in U.

A *function* takes zero or more parameters that are members of U and yields a member of U. Here are examples of functions that might be defined when U is the set of all integers.

- A function with no parameters is called a *constant*. We might define function zero with no parameters to be the constant 0.
- We might define successor(n) to be $n+1$. For example, successor$(2) = 3$.
- We might define sum(m, n) to be $m+n$. For example, sum$(5, 7) = 12$.
- We might define largest(a, b, c) to be the largest of a, b and c. For example, largest$(3, 9, 4) = 9$ and largest$(4, 4, 4) = 4$.

Definition 2.2. A *term* is defined as follows.

1. A *variable* is a term. We use single letters such as x and y for variables.
2. If f is a function that takes no parameters then f is a term (standing for a value in U).
3. If f is a function that takes $n > 0$ parameters and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

For example, sum(sum(x, y), successor(z)) is a term.

The meaning of a term should be clear, provided the values of variables are known. Term sum(x, y) stands for the result that function sum yields on parameters (x, y) (the sum of x and y).

2.3 First-order formulas

Definition 2.3. A *first-order formula* is defined as follows.

1. **T** and **F** are first-order formulas.
2. If P is a predicates that takes no parameters then P is a first-order formula.
3. If $t_1, \ldots, t_n$ are terms and P is a predicate that takes $n > 0$ parameters, then $P(t_1, \ldots, t_n)$ is a first-order formula. It is true if $P(v_1, \ldots, v_n)$ is true, where v_1 is the value of term t_1, v_2 is the value of term t_2, etc.
4. If t_1 and t_2 are terms then $t_1 = t_2$ is a first-order formula. (It is true if terms t_1 and t_2 have the same value.)
5. If A and B are first-order formulas and x is a variable then each of the following is a first-order formula.
   (a) (A)
   (b) $\neg A$
   (c) $A \vee B$
   (d) $A \wedge B$
   (e) $\forall x\, A$
   (f) $\exists x\, A$

The meaning of parentheses, **T**, **F**, $\neg$, $\vee$ and $\wedge$ are the same as in propositional logic. Symbols $\forall$ and $\exists$ are called *quantifiers*. You read $\forall x$ as "for all x, and $\exists x$ as "for some x" or "there exists an x". They have the following meanings.

1. $\forall x\, A$ is true of A is true for all values of x in U.
2. $\exists x\, A$ is true if A is true for at least one value of x in U.

By convention, quantifiers have higher precedence than all of the operators $\wedge$, $\vee$, etc.

Examples of first-order formulas are:

1. $P(\text{sum}(x, y))$ says that, if $v = \text{sum}(x, y)$, then $P(v)$ is true. Its value (true or false) depends on the meanings of predicate P and function sum, as well as on the values of variables x and y.

2. $\forall x(\text{greater}(x, x))$ says that $\text{greater}(x, x)$ is true for every value x in U. Using the meaning of $\text{greater}(a, b)$ given above, $\forall x(\text{greater}(x, x))$ is clearly false, since no x can be greater than itself.

3. $\neg\forall x(\text{greater}(x, x))$ says that $\forall x(\text{greater}(x, x))$ is false. That is true.

4. $\exists y(y = \text{sum}(y, y))$ says that there exists a value y where $y = y + y$. That is true since $0 = 0 + 0$.

5. $\forall x(\exists y(\text{greater}(y, x)))$ says that, for every value v of x, first-order formula $\exists y(\text{greater}(y, v))$ is true. That is true. If $v = 100$, then choose $y = 101$, which is larger than 100. If $v = 1000$, choose $y = 1001$. If $v = 1,000,000$, choose $y = 1,000,001$.

6. $\exists y(\forall x(\text{greater}(y, x)))$ says that there exists a value v of y so that $\forall x(\text{greater}(v, x))$. That is false. There is no single value v that is larger than every integer x.

Operators $\rightarrow$, $\leftrightarrow$ and $\equiv$ have the same meanings in first-order logic as in propositional logic.

2.4 Sentences

Example 1 above uses variable x and y, and its value cannot be determined without knowing the values of x and y. It only makes sense if the values of x and y have already been specified. Think of them as similar to global variables in a function definition in a computer program.

The other examples above do not depend on any variable values. They manage their own variables, and are similar to a function definition that only uses local variables.

We say that variable x is *bound* if it occurs inside A in a first-order formula of the form $\forall x\, A$ or $\exists x\, A$.

Definition 2.4. A first-order formula is a *sentence* if all of its variables are bound.

Table 2-1. Some valid equivalences
$\exists x\, P(x) \lor \neg \exists x\, P(x)$
$\forall x\, P(x) \ \land\ \exists y\, Q(y) \ \equiv\ \exists y\, Q(y) \ \land\ \forall x\, P(x)$
$\neg(\forall x\, A) \ \equiv\ \exists x(\neg A)$
$\neg(\exists x\, A) \ \equiv\ \forall x(\neg A)$
$\forall x(A \land B) \ \equiv\ \forall x\, A \land \forall x\, B$
$\forall x\, A \to \exists x\, A$

2.5 Validity

Recall that a propositional formula is *valid* if it is true for all values of the variables that it contains. There is a similar concept of validity for first-order formulas.

Definition 2.5. Suppose that S is a sentence of first-order logic. (That is, it does not contain any unbound variables.) We say that S is *valid* if it is true regardless of the universe of discourse and the meanings of the predicates and functions that it mentions.

One way to get a valid first-order formula is to substitute first-order formulas into a propositional tautology. The following table lists two valid first-order formulas found in that way. Table 2-1 lists a few valid first-order equivalences, the first two of which are examples of substituting a first-order formula into a propositional equivalence.

2.6 Notation

First-order logic notation is usually extended to include common mathematical notation. For example, we write $x > y$ rather than greater(x, y), and $x + y$ rather than sum(x, y). Constants such as 0, 1 and 200 are also usually allowed. Instead of writing even(x), we write "x is even". For example,

$$\forall x(x \text{ is even } \land y \text{ is even } \to x + y \text{ is even})$$

is true. Those notational changes make first-order logic more readable. Review of first-order logic

3 Theorems and Proofs

A *theorem* is any mathematical statement, such as a formula of first-order logic, that has been proved true. When you are about to prove a theorem, you call it theorem as a way of promising that a proof is about to be produced.

3.1 What is a proof?

There are many different precise definitions of a proof. But most mathematicians accept an informal definition: a proof is a clear and unambiguous argument that a mathematical statement is true that any sufficiently knowledgeable person can check. The key is that a reader must be able to check that each step is correct.

Students who are just learning to do proofs make many different kinds of mistakes, but most fall into one of the following two categories.

1. **The student does not check his or her own work.** The reason can vary from lack of time to lack of understanding to fear of failure.

   A student might take a proof of something else from a book or from notes and make some modifications to it, and then *hope* that their modifications have produced a correct proof, without checking whether that is true. The student who has a lack of understanding cannot check the proof. The student who is afraid of failure will not check the proof out of fear that it might turn out to be incorrect.

   Regardless of your reason for not checking your proof, you can be sure that an unchecked proof is incorrect, for the same reasons that an untested computer program does not work. You will need to find a way to motivate yourself to check your proofs carefully.

2. Mathematics relies on precise definitions. When you do a proof, it is essential for you to use definitions wherever appropriate. **Students often get stuck in a proof because they have forgotten to use definitions**. Any time you cannot see how to proceed, ask yourself if using a definition will help. We will do many examples of that.

3.2 Forward proofs

A *forward proof* reasons from what you know to what you can conclude. Each new conclusion relies on prior knowledge or conclusions.

You have probably been taught a different approach in an algebra class. In a *backwards proof*, you write down what you want to show and then perform some manipulations on it, working backwards to a statement that you already know is true.

In this class, all proof are forward. **I expect you to use only forward proofs as well.** At least for this class, put aside the backwards proofs that you have learned in algebra.

3.3 Some definitions

Definition 3.1. Integer n is *even* if there exists an integer m such that $n = 2 \cdot m$. For example, 6 is even because $6 = 2 \cdot 3$.

Definition 3.2. Integer n is *odd* if there exists an integer m such that $n = 2m + 1$. We will also make use of the fact that, for every n, n is odd if and only if n is not even.

Definition 3.3. Integer n is a *perfect square* if there exists an integer m such that $n = m^2$.

Definition 3.4. Real number x is *rational* if there exist integers n and m where $m \neq 0$ such that $x = n/m$.

Note: In what follows, we will use the convention that $2m$ means $2 \cdot m$ and xy means $x \cdot y$.

3.4 Proof techniques

The remaining subsections discuss common ways of proving particular kinds of first-order formulas. The proofs are detailed and step-by-step. Not all proofs in these notes are so detailed, but details are used later in proofs where an intelligent student might get confused.

3.5 Proving $A \to B$

To prove $A \to B$, assume that A is true and show that B is true.

Theorem 3.1. If n is even then n^2 is even.

Proof.

1. Suppose that n is even.

Known variables:	n
Know:	n is even.
Goal:	n^2 is even.

2. By the definition of an even integer, there exists an integer m such that $n = 2m$.

Known variables:	n, m
Know:	n is even.
Know:	$n = 2m$.
Goal:	n^2 is even.

3. Since $n = 2m$, $n^2 = (2m)^2 = 4m^2 = 2(2m^2)$.

Known variables:	n, m
Know:	n is even.
Know:	$n = 2m$.
Know:	$n^2 = 2(2m^2)$.
Goal:	n^2 is even.

4. So $n^2 = 2(x)$ where $x = 2m^2$. Using the definition of an even number again, n^2 is even.

◇————————————————————◇

Theorem 3.2. If n and m are perfect squares then nm is a perfect square.

Proof.

1. Suppose that n and m are perfect squares.

Known variables:	n, m
Know:	n is a perfect square.
Know:	m is a perfect square.
Goal:	nm is a perfect square.

2. By the definition of a perfect square, there exist integers x and y such that $n = x^2$ and $m = y^2$.

Known variables:	n, m, x, y
Know:	$n = x^2$.
Know:	$m = y^2$.
Goal:	nm is a perfect square.

3. Replacing n by x^2 and m by y^2, $nm = x^2y^2 = (xy)^2$.

Known variables:	n, m, x, y
Know:	$n = x^2$.
Know:	$m = y^2$.
Know:	$nm = (xy)^2$.
Goal:	nm is a perfect square.

4. So $nm = z^2$ where $z = xy$. Using the definition of a perfect square again, nm is perfect square.

$\diamond$————————————————————$\diamond$

3.5.1 Using the contrapositive

You can prove any theorem by proving an equivalent mathematical statement. For example, you can prove $A \to B$ by proving equivalent formula

$\neg B \to \neg A$, which is called the *contrapositive* of $A \to B$. Here is an example.

Theorem 3.3. Suppose n is an integer. If $3n+2$ is odd, then n is odd.

Proof. We prove the contrapositive: If n is not odd then $3n+2$ is not odd.

1. We know that an integer x is even if and only if x is not odd. So what we want to prove is equivalent to: If n is even then $3n+2$ is even.

Known variables:	n
Goal:	If n is even then $3n+2$ is even.

2. Suppose that n is even.

Known variables:	n
Know:	n is even.
Goal:	$3n+2$ is even.

3. By the definition of an even integer, there exists an integer m such that $n = 2m$.

Known variables:	n, m
Know:	$n = 2m$.
Goal:	$3n+2$ is even.

4. $3n+2 = 3(2m)+2 = 6m+2 = 2(3m+1)$.

Known variables:	n, m
Know:	$n = 2m$.
Know:	$3n+2 = 2(3m+1)$.
Goal:	$3n+2$ is even.

5. Using the definition of an even integer again, $3n+2$ is even because $3n+2 = 2z$ where $z = 3m+1$.

$\diamond$ ——— $\diamond$

3.6 Proving and using $A \wedge B$

To prove $A \wedge B$, prove A and prove B.

If you know that $A \wedge B$ is true, then you know that A is true and you know that B is true.

3.7 Proving and using $\neg(A)$

To prove $\neg(A)$, you typically use DeMorgan's laws and the laws for negating quantified formulas to push the negation inward. For example, to prove $\neg(A \wedge B)$, you prove equivalent formula $\neg A \vee \neg B$. To prove $\neg(\forall x A)$, you prove equivalent formula $\exists x(\neg A)$.

The same principle applies when you already know $\neg(A)$. For example, if you know $\neg(A \rightarrow B)$, you can conclude equivalent formula $A \wedge \neg B$. You write that down as an additional known fact.

3.8 Proving and using $A \vee B$

To prove $A \vee B$, you usually prove one of the equivalent formulas $\neg A \rightarrow B$ or $\neg B \rightarrow A$.

Suppose that you know that $A \vee B$ is true and you want to use that to show that C is true. That is, you want to show that $A \vee B \rightarrow C$ is true. You typically prove equivalent formula

$$A \rightarrow C \;\wedge\; B \rightarrow C.$$

That is called *proof by cases*. First, you assume that A is true and show that C is true. Next, you assume that B is true and show that C is true. See Section 3.*** below.

3.9 Proving and using $\exists x A$

To prove that something exists, produce it.

Theorem 3.4. There exists an integer n where n is even and n is prime.

Proof. Choose $n = 2$. Notice that n is even and n is prime.

◇————————————————————◇

3.9.1 Using existential knowledge

Sometimes, instead of needing to prove $\exists x P(x)$, you already know $\exists x P(x)$. What do you do? You ask somebody else to give you a value x so that $P(x)$ is true. It is not necessary for you to say how to find x. We will encounter many examples of that.

3.10 Proving $\forall x A$

To prove $\forall x P(x)$, prove $P(x)$ for an *arbitrary* value of x.

That does not mean that you can choose the value of x. Rather, someone else chooses x and you must prove that $P(x)$ is true for that value of x. Think of it as a challenge. You say to someone else, give me any value of x that you like. I will prove that $P(x)$ is true. In mathematics, *arbitrary* always means a value chosen by someone else.

We have actually used this idea above. When the statement of a theorem involves unbound variables, it is assumed to be saying that the statement is true for all values of those variables. Here is the first proof above with the quantifier explicit. The universe of discourse is the set of all integers.

Theorem 3.5. $\forall n$(n is even $\rightarrow n^2$ is even).

Proof.

1. Ask someone else to select an arbitrary integer n. (We cannot assume anything about n except that it belongs to the universe of discourse.) We must prove: (n is even $\rightarrow n^2$ is even) for that n.

Known variables:	n
Goal:	n is even $\rightarrow n^2$ is even.

2. Suppose that n is even.

Known variables:	n
Know:	n is even.
Goal:	n^2 is even.

3. By the definition of an even integer, there exists an integer m such that $n = 2m$.

Known variables:	n
Know:	$\exists m(n = 2m)$.
Goal:	n^2 is even.

4. Ask someone else to provide the integer m that is asserted to exist.

Known variables:	n, m
Know:	$n = 2m$.
Goal:	n^2 is even.

5. Since $n = 2m$, $n^2 = (2m)^2 = 4m^2 = 2(2m^2)$.

Known variables:	n, m
Know:	$n = 2m$.
Know:	$n^2 = 2(2m^2)$.
Goal:	n^2 is even.

6. So $n = 2(x)$ where $x = 2m^2$. Using the definition of an even number again, n is even.

3.10.1 Proof by contradiction

You can prove any theorem by proving an equivalent theorem. We have seen propositional tautology

$$P \equiv (\neg P \rightarrow \mathbf{F}).$$

That is, to prove P, assume that P is false and prove that $\mathbf{F}$ is true. That is called *proof by contradiction*. Let's use proof by contradition to reprove a theorem that we proved above.

Theorem 3.6. For every integer n, if $3n + 2$ is odd, then n is odd.

Proof.

1. Reasoning by contradiction, we can assume the theorem is false and prove **F**. That is:

Know:	$\neg\forall n(3n + 2 \text{ is odd } \rightarrow n \text{ is odd})$.
Goal:	**F**.

2. We can push the negation across the quantifier using valid formula $\neg\forall xA \equiv \exists x(\neg A))$.

Know:	$\exists n(\neg(3n + 2 \text{ is odd } \rightarrow n \text{ is odd}))$.
Goal:	**F**.

3. Now use the tautology that $\neg(P \rightarrow Q) \equiv P \wedge \neg Q)$.

Know:	$\exists n(3n + 2 \text{ is odd } \wedge n \text{ is even})$.
Goal:	**F**.

4. Ask somebody else to select an integer n such that $3n + 2$ is odd and n is even.

Known variables:	n
Know:	$3n + 2$ is odd.
Know:	n is even.
Goal:	**F**.

5. By the definition of an even integer, saying that n is even is equivalent to saying that there exists an integer m such that $n = 2m$. (Existential information is useful because it allows you to get something in hand, as is done in the next step. So you often want to exploit existential information.)

Known variables:	n
Know:	$3n + 2$ is odd.
Know:	$\exists m(n = 2m)$.
Goal:	**F**.

6. Since we know that an integer m exists such that $n = 2m$, we can ask somebody else to give us such an m. Let's do that.

Known variables:	n, m
Know:	$3n + 2$ is odd.
Know:	$n = 2m$.
Goal:	**F**.

7. Since we know that $n = 2m$, it seems reasonable to substitute $2m$ for n in expression $3n + 2$ to see what we get. Doing that gives

$$3n + 2 = 3(2m) + 2 = 6m + 2 = 2(3m + 1).$$

So $3n + 2$ is even. Recording that:

Known variables:	n, m
Know:	$3n + 2$ is odd.
Know:	$n = 2m$.
Know:	$3n + 2$ is even.
Goal:	**F**.

8. But $3n + 2$ cannot be both even and odd. Formula ($3n + 2$ is odd $\wedge$ $3n + 2$ is even) is equivalent to **F**. So we have concluded that **F** is true and we are done.

◇————————————————————————◇

3.11 Proving $\forall x(\exists y A)$

It is common to encounter theorems whose general form is $\forall x(\exists y P(x, y))$. The proof usually involves finding an algorithm. For any x, the algorithm must find a y so that $P(x, y)$ is true. Here is an example.

Theorem 3.7. For all real numbers x and y, if x and y are both rational numbers then $x + y$ is also a rational number.

Proof.

1. Ask someone else to select arbitrary real numbers of x and y.

Known variables:	x, y
Goal:	If x and y are rational then $x + y$ is rational.

2. Assume that x and y are rational.

Known variables:	x, y
Know:	x is rational.
Know:	y is rational.
Goal:	$x + y$ is rational.

3. Our knowledge involves the term *rational*. We need to know what that means. From the definition of a rational number, there must exist integers a and b where $b \neq 0$ and $x = a/b$; and there must exist integers c and d where $d \neq 0$ and $y = c/d$.

Known variables:	x, y, a, b, c, d
Know:	a, b, c and d are integers.
Know:	$b \neq 0$.
Know:	$d \neq 0$.
Know:	$x = a/b$.
Know:	$y = c/d$.
Goal:	$x + y$ is rational.

4. Since the goal is to show that $x + y$ is rational, let's replace x by a/b and replace y by c/d in expression $x + y$.

$$x + y = a/b + c/d = ad/bd + bc/bd = (ad + bc)/bd.$$

Known variables:	x, y, a, b, c, d
Know:	a, b, c and d are integers.
Know:	$b \neq 0$.
Know:	$d \neq 0$.
Know:	$x = a/b$.
Know:	$y = c/d$.
Know:	$x + y = (ad + bc)/bd$.
Goal:	$x + y$ is rational.

5. But we have shown that $x + y$ is the ratio of integers $ad + bc$ and bd. Since neither b nor d is 0, bd cannot be 0. So $x + y$ is rational, by the definition of a rational number.

◇——————————————————————◇

3.12 Proving $A \equiv B$

There are two commonly used ways of proving $A \equiv B$.

3.12.1 Using direct equivalences

You can treat $\equiv$ in a way similar to the way you treat $=$ in algebraic equations, performing equivalence-preserving manipulations. Let's use that approach to prove the law of the contrapositive.

Theorem 3.8. $P \to Q \ \equiv \ \neg Q \to \neg P$.

Proof.

$$\begin{array}{rcll} \neg Q \to \neg P & \equiv & \neg(\neg Q) \vee \neg P & \text{(defn of } \to) \\ & \equiv & Q \vee \neg P & \text{(double negation)} \\ & \equiv & \neg P \vee Q & \text{(commutative law of } \vee) \\ & \equiv & P \to Q & \text{(defn of } \to) \end{array}$$

3.12.2 Proving two implications

Sometimes it is preferable to use the definition of $P \equiv Q$, namely $P \to Q \wedge Q \to P$.

Theorem 3.9. For every integer n, n is odd if and only if n^2 is odd.

Proof.

1. It suffices to prove

$$\forall n(n \text{ is odd } \to n^2 \text{ is odd } \wedge \ n^2 \text{ is odd } \to n \text{ is odd}).$$

That gives two goals. We use tautology $\forall x(A \wedge B) \equiv \forall x A \wedge \forall x B$ and change the variable names so that we can look at the two parts separately without variables from one interfering with the other.

Goal (1):	$\forall n(n$ is odd $\rightarrow n^2$ is odd).
Goal (2):	$\forall m(m^2$ is odd $\rightarrow m$ is odd).

2. Ask someone else to choose arbitrary values of m and n.

Known variables:	n, m
Goal (1):	n is odd $\rightarrow n^2$ is odd.
Goal (2):	m^2 is odd $\rightarrow m$ is odd.

3. Goal (2) is equivalent to its contrapositive, m is even $\rightarrow m^2$ is even. We proved that as Theorem 3.1. That only leaves Goal (1). (We know goal (2), but we can always discard known things to simplify.)

Known variables:	n
Goal (1):	n is odd $\rightarrow n^2$ is odd.

4. To prove Goal (1), assume that n is odd.

Known variables:	n
Know:	n is odd.
Goal (1):	n^2 is odd.

5. Since n is odd, there exists an integer k so that $n = 2k + 1$.

Known variables:	n
Know:	$\exists k(n = 2k + 1)$.
Goal (1):	n^2 is odd.

6. Ask someone else to provide a value k such that $n = 2k + 1$.

Known variables:	n, k
Know:	$n = 2k + 1$.
Goal (1):	n^2 is odd.

7. Since $n = 2k + 1$,

$$n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1.$$

Since $n^2 = 2z + 1$ for $z = 2k^2 + 2k$, it is evident that n^2 is odd.

$\diamondsuit$——————————————————————$\diamondsuit$

3.13 Proof by cases

Proof by cases involves proving two or more implications. You must be careful that assumptions made during one of those cases are not still in place when proving another one. Think of this is similar to calling a function in a program. Each time a function is called, a new frame is created, so that calling $f(3)$ does not interfere with a later call to $f(4)$.

Theorem 3.10. For every integer n, $n^2 \geq n$.

Proof.

1. Ask someone to select an arbitrary integer n.

Known variables:	n
Know:	n is an integer
Goal:	$n^2 \geq n$.

2. Let's break proving the goal into three cases: $n = 0$, $n > 0$ and $n < 0$.

Known variables:	n
Know:	n is an integer
Goal (1):	$n = 0 \to n^2 \geq n$.
Goal (2):	$n > 0 \to n^2 \geq n$.
Goal (3):	$n < 0 \to n^2 \geq n$.
Goal (4):	$n^2 \geq n$.

3. Goal (1) is clearly true since $0^2 \geq 0$. Let's record it among the known facts.

Known variables:	n
Know:	n is an integer
Know (1):	$n = 0 \rightarrow n^2 \geq n.$
Goal (2):	$n > 0 \rightarrow n^2 \geq n.$
Goal (3):	$n < 0 \rightarrow n^2 \geq n.$
Goal (4):	$n^2 \geq n.$

4. Goal (2) is an implication, so we should assume that $n > 0$ and prove that $n^2 \geq n$. But let's prove that as a separate subproof. Knowledge and goals that are local to the proof of goal (2) is numbered 2.1, 2.2, etc., and they can only be used to establish goal (2).

Known variables:	n
Know:	n is an integer
Know (1):	$n = 0 \rightarrow n^2 \geq n.$
Goal (2):	$n > 0 \rightarrow n^2 \geq n.$
Goal (3):	$n < 0 \rightarrow n^2 \geq n.$
Goal (4):	$n^2 \geq n.$
Know (2.1):	$n > 0$
Goal (2.1):	$n^2 \geq n$

5. Since $n > 0$ is an integer, it must be the case that $n \geq 1$.

Known variables:	n
Know:	n is an integer
Know (1):	$n = 0 \rightarrow n^2 \geq n.$
Goal (2):	$n > 0 \rightarrow n^2 \geq n.$
Goal (3):	$n < 0 \rightarrow n^2 \geq n.$
Goal (4):	$n^2 \geq n.$
Know (2.1):	$n \geq 1$
Goal (2.1):	$n^2 \geq n$

Multiplying both sides of fact (2.1) by n preserves the inequality because $n > 0$. That gives $n \cdot n \geq n \cdot 1$, or equivalently, $n^2 \geq n$.

Known variables:	n
Know:	n is an integer
Know (1):	$n = 0 \rightarrow n^2 \geq n.$
Goal (2):	$n > 0 \rightarrow n^2 \geq n.$
Goal (3):	$n < 0 \rightarrow n^2 \geq n.$
Goal (4):	$n^2 \geq n.$
Know (2.1):	$n \geq 1$
Know (2.2):	$n^2 \geq n$
Goal (2.1):	$n^2 \geq n$

6. We have succeeded in proving goal (2). Notice that fact (2.2) cannot be used to establish goal (4) since it depends on the assumption that $n > 0$.

   We can move goal (2) into our knowledge. But we must also throw out parts that were local to the proof of goal (2).

Known variables:	n
Know:	n is an integer
Know (1):	$n = 0 \rightarrow n^2 \geq n.$
Know (2):	$n > 0 \rightarrow n^2 \geq n.$
Goal (3):	$n < 0 \rightarrow n^2 \geq n.$
Goal (4):	$n^2 \geq n.$

7. Now we need to prove goal (3). Assume that $n < 0$. But the square of any number is nonnegative. It follows that $n^2 \geq 0 > n$, and we can move goal (3) into what we know.

Known variables:	n
Know:	n is an integer
Know (1):	$n = 0 \rightarrow n^2 \geq n.$
Know (2):	$n > 0 \rightarrow n^2 \geq n.$
Know (3):	$n < 0 \rightarrow n^2 \geq n.$
Goal (4):	$n^2 \geq n.$

8. Propositional formula

$$((P \to S) \wedge (Q \to S) \wedge (R \to S)) \to ((P \vee Q \vee R) \to S)$$

is a tautology. That means known facts (1), (2) and (3) imply

$$(n = 0 \vee n > 0 \vee n < 0) \to n^2 \geq n.$$

But we know that $(n = 0 \vee n > 0 \vee n < 0)$ is true, and $\mathbf{T} \to S$ is equivalent to S. So we have demonstrated goal (4).

◇————————————————————————————————◇

That is a long proof, and few people would show it in such detail. Here is the same proof in a more compact form. It is up to you to keep track of what is going on.

Theorem 3.10. For every integer n, $n^2 \geq n$.

Proof. The proof is by cases ($n = 0$, $n > 0$ and $n < 0$).

Case 1 ($n = 0$). Then $n^2 \geq n$ because $0^2 \geq 0$.

Case 2 ($n > 0$). The smallest positive integer is 1, so $n > 0$ implies $n \geq 1$. Multiplying both sides of inequality $n \geq 1$ by positive number n gives $n^2 \geq n$.

Case 3 ($n < 0$). $n^2 \geq 0$ for all numbers n. Since, in this case, n is negative, clearly $n^2 \geq n$.

◇————————————————————————————————◇

Theorems and proofs

4 Mathematical Foundations

4.1 Sets

You should have seen sets before. This is review.

Definition 4.1. A *set* is an unordered collection of things without repetitions. The things in set S are called the *members* of S.

Definition 4.2. A *set enumeration* is one way to describe a set, by writing the members of the set in braces, separated by commas. For example, $\{2, 5, 9\}$ is a set of three integers.

4.1.1 Finite and infinite sets

It is possible to list the members of a *finite* set. But some sets, such as the set of all positive integers, have infinitely many members. Here are a few common infinite sets.

$\mathcal{N}$	$\{0, 1, 2, 3, \ldots\}$
$\mathcal{Z}$	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathcal{R}$	the set of all real numbers

4.1.2 Set comprehensions

A *set comprehension* is a way to describe the set of all values that have a certain property. Notation

$$\{x \mid p(x)\}$$

stands for the set of all values x such that $p(x)$ is true and notation

$$\{f(x) \mid p(x)\}$$

stands for the set of all values $f(x)$ such that $p(x)$ is true. Notation

$$\{x \in S \mid p(x)\}$$

is shorthand for $\{x \mid x \in S \wedge p(x)\}$ Here are some examples.

Set	**Description**
$\{x \mid x \in \mathcal{R} \ \wedge \ x^2 - 2x + 1 = 0\}$	$\{-1,\ 1\}$
$\{x \in \mathcal{R} \mid x^2 - 2x + 1 = 0\}$	$\{-1,\ 1\}$
$\{x \mid x$ is an even positive integer$\}$	$\{2, 4, 6, \ldots\}$
$\{x^2 \mid x$ is an even positive integer$\}$	$\{4, 16, 36, \ldots\}$

4.1.3 Set notation and operations

Table 4-1 defines notation for sets.

4.1.4 Identities for sets

Table 4-2 list some identities are easy to establish.

4.1.5 Sets of sets

The members of sets can be sets. For example, if $S = \{\{1, 2, 3\}, \{2, 4, 6\}\}$ then $|S| = 2$, since S has exactly two members, $\{1,\ 2,\ 3\}$ and $\{2,\ 4,\ 6\}$.

Do not confuse $\in$ with $\subseteq$. If $S = \{\{1, 2, 3\}, \{2, 4, 6\}\}$ then

$\{1, 2, 3\} \in S$

$\{1, 2, 3\} \not\subseteq S$

$3 \notin S$

Notice that $\{\} \neq \{\{\}\}$. $|\{\}| = 0$ but $|\{\{\}\}| = 1$ since $\{\{\}\}$ has one member, the empty set.

4.2 Alphabets and strings

Definition 4.3. An *alphabet* is a finite, nonempty set whose members we call *symbols*.

We will usually want to use small alphabets such as $\{a,\ b\}$ or $\{a,\ b,\ c\}$, where symbols a, b and c stand for themselves (letters of an alphabet).

Table 4-1	
Notation	**Meaning**
$\|S\|$	The *cardinality* (size) of S, when S is a finite set.
$\{\}$	The empty set, which has no members
$x \in S$	True if x is a member of set S. For example, $2 \in \{1, 2, 3, 4\}$
$x \notin S$	$\neg(x \in S)$
$S \cup T$	$\{x \mid x \in S \vee x \in T\}$. For example, $\{2, 5, 6\} \cup \{2, 3, 7\} = \{2, 3, 5, 6, 7\}$. This is called the *union* of sets S and T.
$S \cap T$	$\{x \mid x \in S \wedge x \in T\}$. For example, $\{2, 5, 6\} \cup \{2, 3, 7\} = \{2\}$. This is called the *intersection* of sets S and T.
$S - T$	$\{x \mid x \in S \wedge x \notin T\}$. For example, $\{2, 5, 6\} - \{2, 3, 7\} = \{5, 6\}$. This is called the *difference* of sets S and T.
$\overline{S}$	$U - S$, where U is the universe of discourse. This is called the *complement* of S.
$S \times T$	$\{(x, y) \mid x \in S \wedge y \in T\}$. For example, $\{2, 3\} \times \{5, 6\} = \{(2,5), (2,6), (3,5), (3,6)\}$. This is called the *cartesian product* of S and T.
$S \subseteq T$	This is true if $\forall x(x \in S \rightarrow x \in T)$. For example, $\{2, 4, 6\} \subseteq \{1, 2, 3, 4, 5, 6\}$. Notice that $\{2, 4, 6\} \subseteq \{2, 4, 6\}$. $S \subseteq T$ is read "S is a subset of T."
$S = T$	S and T are the same set if $S \subseteq T$ and $T \subseteq S$. That is, S and T have exactly the same members.

Table 4-2
Some Set Identities
$A \cup \{\} = A$
$A \cap \{\} = \{\}$
$\overline{\overline{A}} = A$
$A \cup B = B \cup A$
$A \cap B = B \cap A$
$A \cup (B \cup C) = (A \cup B) \cup C$
$A \cap (B \cap C) = (A \cap B) \cap C$
$\overline{A \cup B} = \overline{A} \cap \overline{B}$
$\overline{A \cap B} = \overline{A} \cup \overline{B}$
$A - B = A \cap \overline{B}.$
$A \cup (A \cap B) = A$
$A \cap (A \cup B) = A$

It is conventional to call an alphabet Σ (upper case Greek letter sigma, indicating *symbol*).

Definition 4.4. If Σ is an alphabet, then a *string over Σ* is a finite sequence members of Σ. (In a sequence, order matters and there can be repetitions.)

I will write strings in double-quotes. For example, if $\Sigma = \{a, b, c\}$ then "aab" and "$ccccc$" are two strings over Σ.

A fundamental operations on strings is *concatenation*, where $s \cdot t$ indicates s followed by t. For example, "abc" $\cdot$ "aba" = "$abcaba$". Just as the multiplication symbol is usually unwritten between numbers, we will usually omit the concatenation dot between strings and write st to mean $s \cdot t$.

We will allow concatenation to work with symbols as well as strings. For example, "aab" $\cdot\, a$ = "$aaba$".

When the alphabet is understood or unimportant, we talk about a *string*, leaving the alphabet unstated.

Definition 4.5. If s is a string, then $|s|$ is the length of s (the number of characters in s). For example, $|$"$accb$"$| = 4$ and $|$"b"$| = 1$.

Definition 4.6. We write ε to mean the empty string, "", whose length is 0. (Symbol ε is a variant of Greek letter epsilon. Think of it as e for empty.)

4.2.1 Sets of Strings

Definition 4.7. A set of strings is called a *language*.

Definition 4.8. If Σ is an alphabet, then Σ^* is the set of all strings over Σ. For example, $\{a, b\}^* = \{\varepsilon$, "$a$", "$b$", "$aa$", "$ab$", "$ba$", "$bb$", "$aaa$", $\dots\}$.

4.2.2 Natural numbers as strings

We will use strings as the inputs and outputs of algorithms or programs. But sometimes, we want the inputs and outputs to be integers. That is easy to manage: we write the integers in standard (decimal) notation as string. For example, 25 is treated as string is "25".

4.3 Functions

You should have seen functions before. This is review.

Definition 4.9. If A and B are sets, then a *function with domain A codomain B* associates exactly one value in set B with each value in set A. We write $f : A \to B$ to mean that f is a function with domain A and codomain B.

Definition 4.10. If $f : A \to B$ and $x \in A$, then notation $f(x)$ indicates the member of B that f associates with x. When $f(x) = y$, we say that f *maps* x to y.

For example, suppose that $f : \mathcal{N} \to \mathcal{N}$ is defined by $f(x) = x^2$. Then $f(3) = 9$ and $f(5) = 25$.

4.4 Computational problems

We will look at two kinds of computational problems.

1. A *decision problem* is a problem where the input is a string (over a chosen *input alphabet*) and the output is either 1 (true) or 0 (false). We can also think of the output as yes or no.

   A decision problem can be expressed as a function or as a set of strings (a language). When S is a set of strings, we think of S as the decision problem:

   > **Input.** String x over the input alphabet.
   >
   > **Question.** Is $x \in S$?

   Most of the problems that we look at will be decision problems.

2. A *functional problem* is a problem where the input is a string (over the *input alphabet*) and the output is a string (over the *output alphabet*).

4.5 Types

It is easy to become confused about different types of things. Adjectives or other terms that we define can only be applied to certain types of things. For example, it makes sense to talk about the cardinality of a set, but not the cardinality of a number. The following is a list of some of the types of things that we will use.

Type	**Meaning**
boolean	A boolean value is either true or false. It might equally well be either 1 or 0, or either yes or no.
symbol	A symbol is a member of some alphabet.
string	A string is a (possibly empty) finite sequence of symbols
language	A language is a set of strings. We can think of a language as a decision problem.
function	Generally, our functions will either take a string and yield a boolean value or will take a string and yield a string.
set of languages	A set of languages is called a *class*. We think of a language as a decision problem, and we will identify classes of decision problems that can be solved in particular ways.

Mathematical foundations

5 Finite-State Machines and Regular Languages

This section looks at a simple model of computation for solving decision problems: a finite-state machine, or FSM.

5.1 Intuitive idea of a FSM

Figure 5-1 shows a diagram, called a *transition diagram*, of FSM M_1. Each circle or double-circle is called a *state*. One of the states, marked by an arrow, is called the *start state*. A state with a double circle is called an *accepting state* and a state with a single circle is called a *rejecting state*.

The arrows between states are called *transitions*, and each transition is labeled by a member of the FSM's alphabet Σ (set $\{a, b\}$ for M_1). For each state q and each member c of Σ, there must be exactly one transition going out of q labeled c.

A FSM is used to recognize a language (a decision problem). To "run" a FSM on string s, start in the start state. Read each character, and follow the transition labeled by that character to the next state. On input "$aabab$", M_1 starts in state 1, then hits states 2, 1, 1, 2, 2, ending in state 2.

The end state determines whether the FSM accepts or rejects the string. Since state 2 is a rejecting state, M_1 rejects "$aabab$". It should be easy to see that M_1 accepts strings with an even number of as and rejects strings with an odd number of as.

A FSM M with alphabet Σ *recognizes* the set

$$L(M) = \{s \mid s \in \Sigma \text{ and } M \text{ accepts } s\}.$$

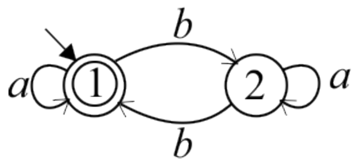


Figure 5-1. Transition diagram of FSM M_1 that recognizes language $\{s \in \{a, b\}^* \mid s \text{ has an even number of } a\text{s}\}$. There are two states. State 1 is the start state. State 1 is an accepting state and state 2 is a rejecting state..

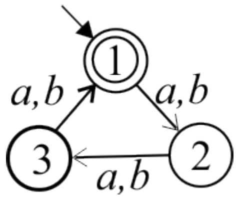


Figure 5-2. Transition diagrams of FSM M_2, which accepts strings whose length is divisible by 3.

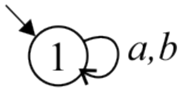


Figure 5-3. Transition diagrams of FSM M_3, which rejects all strings.

For example, $L(M_1) = \{s \mid s \in \{a,b\}^* ands$ has an even number of as$\}$. Figures 5-2 and 5-3 show two finite-state machines M_2 and M_3 with alphabet $\{a,b\}$ where

$$\begin{aligned} L(M_2) &= \{s \mid \ |s| \text{ is divisible by 3}\} \\ L(M_3) &= \{\} \end{aligned}$$

5.2 Designing FSMs

There is a simple and versatile way to design a FSM machine to recognize a selected language L. Associate with each state q the set of strings Set(q) that end on state q. For example, in machine M_2,

$$\begin{aligned} \text{Set}(0) &= \{s \mid \ |s| \equiv 0 \pmod 3\} \\ \text{Set}(1) &= \{s \mid \ |s| \equiv 1 \pmod 3\} \\ \text{Set}(2) &= \{s \mid \ |s| \equiv 2 \pmod 3\} \end{aligned}$$

Your goals in designing a FSM that recognizes language L are:

(a) Start by deciding what the states will be and what Set(q) will be for each state. Make sure that, for each state q, either Set$(q) \subseteq L$ or Set$(q) \subseteq \overline{L}$.

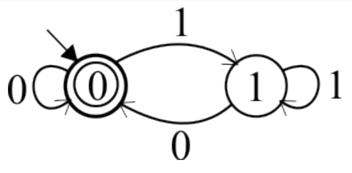


Figure 5-4. A FSM that recognizes even binary numbers. An empty string is treated as 0.

(b) Make q be an accepting state if $\text{Set}(q) \subseteq L$ and make q a rejecting state if $\text{Set}(q) \subseteq \overline{L}$.

(c) Draw transitions so that, if $x \in \text{Set}(q)$ and there is a transition from state q to state q' labeled a, then $x \cdot a \in \text{Set}(q')$.

5.2.1 Example: even binary numbers

Figure 5-4 shows a FSM with alphabet {0,1} that accepts all even binary numbers. For example, it accepts `"10010"` and rejects `"1101"`. $\text{Set}(0) = \{s \in \{0,1\}^* \mid s \text{ is an even binary number}\}$ and $\text{Set}(1) = \{s \in \{0,1\}^* \mid s \text{ is an odd binary number}\}$. The transitions are obvious: adding a 0 to the end of any binary number makes the number even, and adding a 1 to the end makes the number odd.

5.2.2 A FSM recognizing binary numbers that are divisible by 3

Figure 5-5 shows a FSM that recognizes binary numbers that are divisible by 3. For example, it accepts `"1001"` and `"1100"`, since `"1001"` is the binary representation of 9 and `"1100"` is the binary represention of 12. But it rejects `"100"`, the binary representation of 4.

Thinking of binary strings as representing numbers,

$$\begin{aligned}
\text{Set}(0) &= \{n \mid n \equiv 0 \pmod 3\} \\
\text{Set}(1) &= \{n \mid n \equiv 1 \pmod 3\} \\
\text{Set}(2) &= \{n \mid n \equiv 2 \pmod 3\}
\end{aligned}$$

Suppose that m is a binary number that is divisible by 3. Adding a 0 to the end doubles the number, so $m \cdot 0$ is also divisible by 3. Adding a 1 to m doubles m and adds 1. But modular arithmetic tells us that

$$\begin{aligned} m \equiv 0 \pmod 3 &\rightarrow 2m \equiv 0 \pmod 3 \\ &\rightarrow 2m+1 \equiv 1 \pmod 3 \end{aligned}$$

so there is a transition from state 0 to state 1 on symbol 1.

5.2.3 Strings containing at least two as and at most one b.

Figure 5-6 shows a FSM that regognizes language

$$\{w \in \{a, b\}^* \mid w \text{ contains at least two } a\text{s and at most one } b\}.$$

The idea is to keep track of the number of as (up to a maximum of 2) and the number of bs (up to a maximum of 2). That suggests that we need nine states: $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$, $(1, 2)$, $(2, 0)$, $(2, 1)$ and $(2, 2)$, where the first number is the count of as and the second the count of bs, and 2 means at least 2. The accepting states and transitions should be obvious.

5.3 Definition of a FSM and the class of regular languages

The introduction above only shows transition diagrams, and does not adequately say exactly what a FSM is and how to determine the language that it recognizes. This section corrects that with a careful definition of both. The first definition says what a FSM is without saying about what it means to run it on a string.

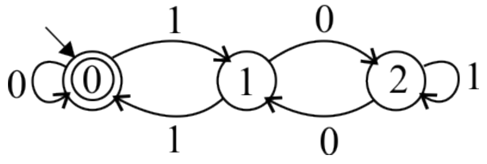


Figure 5-5. A FSM recognizing binary numbers that are divisible by 3. An empty string is treated as 0.

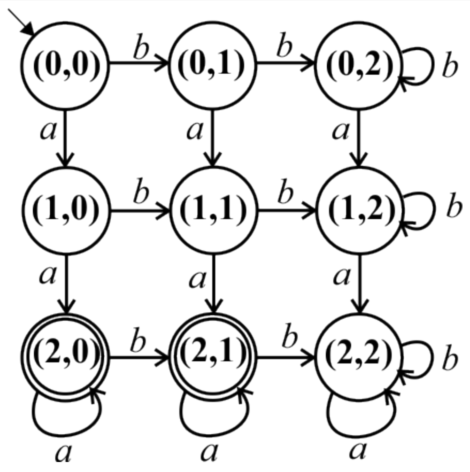


Figure 5-6. A FSM recognizing strings of as and bs with at least two as and at most one b.

5.3.1 Definition of a FSM

Definition 5.1. A *finite-state machine* is a 5-tuple (Σ, Q, q_0, F, δ). That is, it is described by those five parts.

- Σ is the machine's alphabet.
- Q is a finite nonempty set whose members are called *states*.
- $q_0 \in Q$ is called the *start state*.
- $F \subseteq Q$ is the set of *accepting states*.
- $\delta : Q \times \Sigma \to Q$ is called the *transition function*.

Most of that should be clear from the transition diagrams that we have looked at. From state q, if you read symbol a, you go to state $\delta(q, a)$. Notice that, because δ is a function, there must be exactly one state to go to from state q upon reading symbol a.

5.3.2 When does FSM M accept string s?

Consider a FSM $M = (\Sigma, Q, q_0, F, \delta)$.

Definition 5.2. If $q \in Q$ and $x \in \Sigma^*$, then $q : x$ is defined inductively as follows.

1. $q : \varepsilon = q$.
2. If $x = cy$ where $c \in \Sigma$ and $y \in \Sigma^*$ then $q : x = \delta(q, c) : y$.

The idea is that $q : x$ is the state that M reaches if it starts in state q and reads string x. To find that out for string $x = cy$, first find the state $q' = \delta(q, c)$, then finish by finding $q' : y$.

Every FSM M has a language $L(M)$ that it recognizes, and the following definition says what that is.

Definition 5.3. $L(M) = \{x \in \Sigma^* \mid q_0 : x \in F\}$.

That is, M accepts string x if M reaches an accepting state when it is run on x starting in the start state, q_0.

5.3.3 The class of regular languages

Definition 5.4. Language A is *regular* if there exists a FSM M such that $L(M) = A$.

We have see a few regular languages above, including $\{\}$ and the set of binary numbers that are divisible by 3.

5.4 A theorem about $q : x$

Notation $q : x$ satisfies a certain kind of associativity.

Theorem 5.1. $(q : x) : y = q : (xy)$.

Proof. The proof is by induction of the length of x. The introduction to proofs does not cover proof by induction because this is the only such proof that we need. It suffices to

(a) show that $(q:x):y = q:(xy)$ for all q and y when $|x| = 0$, and

(b) show that $(q:x):y = q:(xy)$ for an arbitrary nonempty string x, under the assumption (called the *induction hypothesis*) that $(r:z):y = r:(zy)$ for any state r, string y and string z that is shorter than x.

Case 1 ($|x| = 0$). That is, $x = \varepsilon$. By definition, $q:\varepsilon = q$. So

$$\begin{aligned}(q:x):y &= q:y \\ &= q:(xy)\end{aligned}$$

because, when $x = \varepsilon$, $xy = y$.

Case 2 ($|x| > 0$). A nonempty string x can be broken into $x = cz$ where c is the first symbol of x and z is the rest.

$$\begin{aligned}(q:x):y &= (q:(cz)):y \\ &= (\delta(q,c):z):y && \text{by the definition of } q:(cz) \\ &= \delta(q,c):(zy) && \text{by the induction hypothesis} \\ &= q:(czy) && \text{by the definition of } q:(czy) \\ &= q:(xy) && \text{since } x = cz\end{aligned}$$

5.5 Closure results

A *closure* result tells you that a certain operation does not take you out of a certain set. For example, $\mathcal{Z}$ is *closed under addition* because the sum of two integers is an integer. $\mathcal{Z}$ is also *closed under multiplication*. But $\mathcal{Z}$ is not closed under division, since $1/2$ is not an integer.

The class of regular languages possesses some useful closure results.

Definition 5.5. Suppose that $A \subseteq \Sigma^*$ is a language. The complement $\overline{A}$ of A is $\Sigma^* - A$.

Theorem 5.2. The class of regular languages is closed under complementation. That is, if A is a regular language then $\overline{A}$ is also a regular language. Put another way, for every FSM M, there is another FSM M' where

$L(M') = \overline{L(M)}$. Moreover, there is an algorithm that, given M, finds M'. That is, the proof is constructive.

Proof. Suppose that $M = (\Sigma, Q, q_0, F, \delta)$. Then $M' = (\Sigma, Q, q_0, Q - F, \delta)$. That is, simply convert each accepting state to a rejecting state and each rejecting state to an accepting state.

◇————————————————————————————◇

Theorem 5.3. The class of regular languages is closed under intersection. That is, if A and B are regular languages then $A \cap B$ is also a regular language. Put another way, suppose M_1 and M_2 are FSMs with the same alphabet Σ. There is a FSM M' so that $L(M') = L(M_1) \cap L(M_2)$. That is, M' accepts x if and only if both M_1 and M_2 accept x. Moreover, there is an algorithm that takes parameters M_1 and M_2 and produces M'.

Proof. The idea is to make M' simulate M_1 and M_2 at the same time. For that, we want a state of M' to be an ordered pair holding a state of M_1 and a state M_2. Recall that the cross product $A \times B$ of two sets A and B is $\{(a, b) \mid a \in A \wedge b \in B\}$.

Suppose that $M_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$. and $M_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$. Then $M' = (\Sigma, Q', q'_0, F', \delta')$ where

$$\begin{aligned} Q' &= Q_1 \times Q_2 \\ q'_0 &= (q_{0,1}, q_{0,2}) \\ F' &= F_1 \times F_2 \\ \delta'((r, s), a) &= (\delta_1(r, a), \delta_2(s, a)) \end{aligned}$$

State (r, s) of M' indicates that M_1 is in state r and M_2 is in state s. Transition function δ' runs M_1 and M_2 each one step separately. Notice that the set F' of accepting states of M' contains all states (r, s) where r is an accepting state of M_1 and s is an accepting state of M_2. So M' accepts x if and only if both M_1 and M_2 accept x.

◇————————————————————————————◇

Theorem 5.4. The class of regular languages is closed under union. That is, if A and B are regular languages then $A \cup B$ is also a regular language.

Proof. By DeMorgan's laws for sets,

$$A \cup B = \overline{\overline{A} \cap \overline{B}}.$$

By we know that the class of regular languages is closed under complementatin and intersection. Finite-state machines and regular languages

6 Regular Expressions

This section introduces regular expressions. A regular expression describes a set of strings. The class of languages that can be decribed by regular expressions is exactly the class of regular languages, which Section 5defines to be the class of languages that can be solved by finite-state machines.

6.1 Regular operations

The regular operations are operations on languages. The first regular operation is union ($A \cup B$), which we have already seen. The remaining two regular operations are concatenation and Kleene closure.

Definition 6.1. The *concatenation* $A \cdot B$ of languages A and B is defined by

$$A \cdot B = \{xy \mid x \in A \text{ and } y \in B\}.$$

That is, $A \cdot B$ is the set of all strings that can be formed by writing a member of A followed by a member of B. For example, $\{"aa", "ccb"\} \cdot \{"abc", "bb"\} = \{"aaabc", "aabb", "ccbabc", "ccbbb"\}$.

Definition 6.2. The *Kleene closure* A^* of language A is defined by

$$A^* = \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A \text{ for } i = 1, \ldots, n\}.$$

If $A = \{"a", "bcb"\}$ then $A^* = \{\varepsilon, "a", "bcb", "aa", "abcb", "bcba", "bcbbcb", \ldots\}$. A^* contains the empty string and all strings that can be formed by concatenating members of A together. Notice that $\{\}^* = \{\varepsilon\}$.

Language L is *closed under concatenation* if, whenever x and y are both in L, xy is also in L. Another way to define the Kleene closure of A is as the smallest set of strings that is closed under concatenation and that contains ε and all members of A.

6.2 Regular expressions

A regular expression e over alphabet Σ is an expression whose value is a language $L(e)$ over Σ. Regular expressions have the following forms.

1. A symbol $a \in \Sigma$ is a regular expression. $L(a) = \{"a"\}$.
2. If A and B are regular expressions, then:
   (a) $A \cup B$ is a regular expression. $L(A \cup B) = L(A) \cup L(B)$.
   (b) AB is a regular expression. $L(AB) = L(A) \cdot L(B)$.
   (c) A^* is a regular expression. $L(A^*) = L(A)^*$.

Conventionally, * has highest precedence, followed by concatenation, with $\cup$ having lowest precedence. You can use parentheses to override precedence rules.

We put spaces in some regular expressions to make them more readable.

6.3 Regular expressions and regular languages

We do not have time to prove the following two theorems.

Theorem 6.1. If e is a regular expression then $L(e)$ is a regular language.

Theorem 6.1. If A is a regular language then there exists a regular expression e so that $L(e) = A$.

We have two very different ways to describe the class of regular languages: as languages that are decidable by FSMs and as languages that can be described by regular expressions.

Just because two things are defined differently does not necessarily make them different things.

6.4 Examples of regular expressions

$(ab)^*$	any string over alphabet $\{a, b\}$ that consists of ab repeated zero or more times. $\{\varepsilon$, "ab", "$abab$", "$ababab$", $\dots\}$
a^*b^*	any string over alphabed $\{a, b\}$ that consists of zero or more as followed by zero or more bs. $\{\varepsilon$, "a", "b", "ab", "aab", "$aabb$", $\dots\}$.
$(a \cup b)^*$	all strings over alphabet $\{a, b\}$.
$(a \cup b)^*a(a \cup b)$	all strings over alphabet $\{a, b\}$ whose next-to-last character is a.
$(a \cup b)^*aabb(a \cup b)^*$	all strings over alphabet $\{a, b\}$ that have $aabb$ as a contiguous substring.
$b^*(ab^*a)^*b^*$	all strings over alphabet $\{a, b\}$ that have an even number of as.
$(0 \cup 1(01^*0)^*1)^*$	all binary numbers that are divisible by 3. (This one is difficult and is not obvious. Look at the FSM in Figure 5-5. Starting in state 0, what can the FSM read to get it back to state 0? Certainly, it can read a 0. It can also read a 1, taking it to state 1, then 01^*0 repeated any number of times, then a 1 to get it back to state 0. Those two, getting the FSM from state 0 back to state 0, can be repeated any number of times.

Exercises

Why can't you write a regular expression e so that $L(e)$ is the set of all strings over $\{a, b\}$ that have the same number of a's as bs? Regular expressions

7 Nonregular Languages

7.1 A motivating example

Notation a^n means a string of n consecutive as. For example, $a^1 =$ "a", $a^2 =$ "aa" and $a^3 =$ "aaa". It is easy to design a finite-state machine that solves language

$$L_1 = \{a^m b^n \mid m > 0 \text{ and } n > 0\}.$$

A string s is in L_1 if and only if s consists of some positive number of as followed by a positive number of bs. But suppose that

$$L_2 = \{a^n b^n \mid n > 0\}.$$

Notice that a string s is in L_2 if and only if s consists of some positive number of as followed by *the same number* of bs. $L_2 = \{$"ab", "$aabb$", "$aaabbb$", $\ldots\}$.

Suppose that you want to design a finite state machine M where $L(M) = L_2$. What information does M need to remember? What if M reads a string of n as and the next symbol is a b? M must remember n. If it doesn't, then how will M be able to check whether there are exactly n bs?

So it seems that M must have a state remembering that it has read exactly 1 a, another state remembering that it has read exactly 2 as, another remembering that it has read exactly 3 as, etc., without any limit. But that requires infinitely many states!

Can we conclude that L_2 is not regular? Be careful! Many incorrect "proofs" have been proposed that follow the rough outline: "I can only only see one way to solve this problem. That way does not work. Therefore, this problem is unsolvable." That is nonsense. What if you missed an idea? We need a more careful proof.

7.2 A proof technique

The idea about why L_2 is not regular is sound, but it needs to be presented more carefully. This section illustrates a way to show that a language is not regular (if it really isn't regular), illustrated with L_2.

Theorem 7.1. L_2 is not regular.

Proof.

1. The proof is by contradiction. Suppose that L_2 is regular. We need to derive a contradiction by proving that **F** is true.

Know:	L_2 is regular.
Goal:	**F**.

2. Our knowlede uses term *regular*. By definition, L_2 is regular if an only if there is a FSM M where $L(M) = L_2$.

Know:	There exists a FSM M where $L(M) = L_2$.
Goal:	**F**.

3. When you know that there exists something with a particular property, ask someone else to give you such a thing. So let's ask for M, and suppose the start state of M is q_0.

Known variables:	M, q_0
Know:	$L(M) = L_2$.
Know:	q_0 is the start state of M.
Goal:	**F**.

4. This kind of proof involves a clever idea, and here it is. Our intuitive reasoning above looked at the state that M reaches after reading each of a^1, a^2, a^3, etc. So let's think about those states. In fact, since we have M in hand, we can do an experiment where we run M on each of a^1, a^2, a^3, a^4, etc. For each one, let's write the state that M reaches. That gives a table that *might* start out looking like this.

Input x	**State** $q_0 : x$ **reached**
"a"	2
"aa"	6
"aaa"	3
"$aaaa$"	9
...	...

But M only has finitely many states. By the pigeonhole principle, as you expand the table for longer and longer strings of as, there must come a point where a state is repeated. Suppose that strings a^i and a^j take M to the same state q.

Input x	**State** $q_0 : x$ **reached**
$\dots$	$\dots$
a^i	q
$\dots$	$\dots$
a^j	q
$\dots$	$\dots$

The experiment shows that $q_0 : a^i = q_0 : a^j = q$.

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_2$.
Know:	q_0 is the start state of M.
Know:	$q_0 : a^i = q$.
Know:	$q_0 : a^j = q$.
Know:	$i < j$.
Goal:	**F**.

5. Now comes a second clever trick. We have seen that M forgets the difference between a^i and a^j, since the only thing M can remember is the state that it is in. What if i bs come next? On input a^ib^i, M should answer yes. But on input a^jb^i, M should answer no.

   Define $q' = q : b^i$. Recalling that $q = q_0 : a^i$ and $q = q_0 : a^j$,

$$\begin{array}{rcll} q' & = & q : b^i & \\ & = & (q_0 : a^i) : b^i & \\ & = & q_0 : a^ib^i & \text{(by Theorem 5.1)} \\ q' & = & q : b^i & \\ & = & (q_0 : a^j) : b^i & \\ & = & q_0 : a^jb^i & \text{(by Theorem 5.1)} \end{array}$$

   So M reaches the same state q' on input a^ib^i as on input a^jb^i.

Suppose that q' is an accepting state. Then M correctly accepts $a^i b^i$ but incorrectly accepts $a^j b^i$.

Suppose that q' is a rejecting state. Then M correctly rejects $a^j b^i$ but incorrectly rejects $a^i b^i$.

No matter what, M does not correctly solve language L_2.

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_2$.
Know:	q_0 is the start state of M.
Know:	$q_0 : a^i = q$.
Know:	$q_0 : a^j = q$.
Know:	$i < j$.
Know:	$L(M) \neq L_2$.
Goal:	**F**.

6. That gives us the contradiction: $(L(M) = L_2) \wedge (L(M) \neq L_2) \;\equiv\; \mathbf{F}$.

◇————————————————————◇

The above proof is actually quite constructive. Suppose that Archibald says he can produce a FSM M that solves L_2. Ask for it. Upon receiving M from him, apply the above idea. You find a string on which M gets the wrong answer. Sending that string to Archibald provides him with an irrefutable reason to believe that he was mistaken.

7.3 Another example

Suppose

$$L_3 = \{a^n \mid n \text{ is a perfect square}\}.$$

Theorem 7.2. L_3 is not regular.

Proof.

1. The proof is by contradiction. Suppose that L_3 is regular. We need to derive a contradiction by proving that **F** is true.

Know:	L_3 is regular.
Goal:	**F**.

2. By definition, L_3 is regular if an only if there is a FSM M where $L(M) = L_3$.

Know:	There exists a FSM M where $L(M) = L_3$.
Goal:	**F**.

3. Ask someone else to give you a FSM M where $L(M) = L_3$. Suppose the start state of M is q_0.

Known variables:	M, q_0
Know:	$L(M) = L_3$.
Know:	q_0 is the start state of M.
Goal:	**F**.

4. To employ the first clever idea, we need to find an infinite sequence of strings to try M on. The requirement is that M cannot afford to forget the difference between any two of those infinitely many strings; it needs to stop in a different state for each of them. Finding that sequence is the part of this kind of proof that requires the most thought.

   A sequence of strings that does the job is a^1, a^4, a^9, a^{16}, etc.; that is, a^{1^2}, a^{2^2}, a^{3^2}, a^{4^2}, etc. We have M in hand, and we can do an experiment where we run M on each of those strings. The table *might* start out looking like this.

Input x	**State** $q_0 : x$ **reached**
a^{1^2}	8
a^{2^2}	1
a^{3^2}	14
a^{4^2}	6
...	...

Since M only has finitely many states, but there are infinitely many strings in the sequence, the right-hand column must eventually contain a repetition. Suppose that inputs a^{i^2} and a^{j^2} stop on the same state, q.

Input x	**State** $q_0 : x$ **reached**
$\ldots$	$\ldots$
a^{i^2}	q
$\ldots$	$\ldots$
a^{j^2}	q
$\ldots$	$\ldots$

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_2$.
Know:	q_0 is the start state of M.
Know:	$q_0 : a^{i^2} = q$.
Know:	$q_0 : a^{j^2} = q$.
Know:	$i < j$.
Goal:	**F**.

5. With the second clever trick, we must show that the first clever trick was done correctly. We have seen that M forgets the difference between a^{i^2} and a^{j^2}, since the only thing M can remember is the state that it is in. Our goal is to find *one* string r where M should accept $a^{i^2}r$ but M should reject $a^{j^2}r$. A string r that does the job is $r = a^{2i+1}$. Notice that

$$\begin{array}{rcl} a^{i^2}r & = & a^{i^2}a^{2i+1} \\ & = & a^{i^2+2i+1} \\ & = & a^{(i+1)^2} \end{array}$$

So $a^{i^2}r \in L_3$. But

$$\begin{array}{rcl} a^{j^2}r & = & a^{j^2}a^{2i+1} \\ & = & a^{j^2+2i+1} \end{array}$$

But $i < j$, so

$$\begin{aligned} j^2 &< j^2 + 2i + 1 \\ &< j^2 + 2j + 1 \\ &= (j+1)^2 \end{aligned}$$

Since there are no perfect squares between j^2 and $(j+1)^2$, $j^2 + 2i + 1$ cannot be a perfect square. That means $a^{j^2}r \notin L_3$.

Recall that M stops in the same state, q, on input a^{i^2} as on input a^{j^2}. Therefore, it stops on the same state $q' = q : r$ on input $a^{i^2}r$ as on input $a^{j^2}r$.

If q' is an accepting state, then M correctly accepts $a^{i^2}r$ but incorrectly accepts $a^{j^2}r$.

If q' is an rejecting state, then M correctly rejects $a^{j^2}r$ but incorrectly rejects $a^{i^2}r$.

No matter what, there is an input on which M gives the wrong answer. So $L(M) \neq L_3$.

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_3$.
Know:	$L(M) \neq L_3$.
Know:	q_0 is the start state of M.
Know:	$q_0 : a^i = q$.
Know:	$q_0 : a^j = q$.
Know:	$i < j$.
Goal:	**F**.

6. That gives us the contradiction: $(L(M) = L_3) \wedge (L(M) \neq L_3) \equiv \mathbf{F}$.

◇————————————————————◇

7.4 Another example

Suppose

$$L_4 = \{ww \mid w \in \{a,b\}^*\}.$$

Strings in L_4 include "aa", "$abab$", "$aabbbaabbb$" and "$bbaabbaa$", among infinitely many others.

Theorem 7.3. L_4 is not regular.

Proof.

1. As before, the proof is by contradiction. Suppose that L_4 is regular. We need to derive a contradiction by proving that **F** is true.

Know:	L_4 is regular.
Goal:	**F**.

2. By definition, L_4 is regular if an only if there is a FSM M where $L(M) = L_4$.

Know:	There exists a FSM M where $L(M) = L_4$.
Goal:	**F**.

3. Ask someone else to provide us with a FSM M where $L(M) = L_4$. Suppose the start state of M is q_0.

Known variables:	M, q_0
Know:	$L(M) = L_4$.
Know:	q_0 is the start state of M.
Goal:	**F**.

4. We need to find an infinite sequence of strings to try M on, where M cannot afford to forget the difference between any two of those strings. A sequence that works is a^1b, a^2b, a^3b, etc. Let's try running M on those strings and write down the state that M reaches for each of them. The experiment *might* yield the following.

Input x	**State** $q_0 : x$ **reached**
"ab"	1
"aab"	2
"$aaab$"	3
"$aaaab$"	4
...	...

But M only has finitely many states. As you expand the table for longer and longer strings, there must come a point where a state is repeated. Suppose that strings a^ib and a^jb take M to the same state q.

Input x	**State** $q_0 : x$ **reached**
$\ldots$	$\ldots$
a^ib	q
$\ldots$	$\ldots$
a^jb	q
$\ldots$	$\ldots$

The experiment shows that $q_0 : a^ib = q_0 : a^jb = q$.

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_4$.
Know:	q_0 is the start state of M.
Know:	$q_0 : a^ib = q$.
Know:	$q_0 : a^jb = q$.
Know:	$i < j$.
Goal:	**F**.

5. M forgets the difference between a^ib and a^jb, since the only thing M can remember is the state that it is in. What if string $r = a^ib$ comes next? On input a^iba^ib, M should answer yes, since $a^iba^ib = ww$ where $w = a^ib$. But on input a^jba^ib, M should answer no, since there does not exist any string w where $a^jba^ib = ww$. But M reaches the same state $q' = q : r$ on a^iba^ib as on a^jba^ib. If q' is an accepting state, then M incorrectly accepts a^jba^ib. If q' is a rejecting state, then M incorrectly rejects a^iba^ib. So M does not solve L_4.

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_4$.
Know:	q_0 is the start state of M.
Know:	$q_0 : a^i = q$.
Know:	$q_0 : a^j = q$.
Know:	$i < j$.
Know:	$L(M) \neq L_4$.
Goal:	**F**.

6. That gives us the contradiction.

◇————————————————————◇

7.5 A common mistake

Suppose $L_5 = \{a^n a^n \mid n > 0\}$. Let's try to prove the following.

Claim. L_5 is not regular.

"Proof."

1. The proof is by contradiction. Suppose that L_5 is regular. We need to derive a contradiction by proving that **F** is true.

Know:	L_5 is regular.
Goal:	**F**.

2. By definition, L_5 is regular if an only if there is a FSM M where $L(M) = L_5$.

Know:	There exists a FSM M where $L(M) = L_5$.
Goal:	**F**.

3. Ask someone else to give such you a FSM M where $L(M) = L_5$, and suppose the start state of M is q_0.

Known variables:	M, q_0
Know:	$L(M) = L_5$.
Know:	q_0 is the start state of M.
Goal:	**F**.

4. Do an experiment using M. Run it on sequence of strings a^1, a^2, a^3, etc. and record the state reached for each string. Continue until there is a state (q) has been written twice, which must happen because M has finitely many states.

Input x	**State** $q_0 : x$ **reached**
...	...
a^i	q
...	...
a^j	q
...	...

Known variables:	M, q_0, q, i, j
Know:	$L(M) = L_2$
Know:	q_0 is the start state of M
Know:	$q_0 : a^i = q$
Know:	$q_0 : a^j = q$
Know:	$i < j$
Goal:	**F**

5. Now we need to find a string r so that $a^i r \in L_5$ but $a^j r \in L_5$. Choose $r = a^i$.

   Notice that $a^i r = a^i a^i$ and that is in L_5 from the definition of L_5.

   Notice that $a^j r = a^j a^i$. But that does not have the form $a^n a^n$ so $a^j r \notin L_5$.

   As before, that leads to a contradiction.

$\diamondsuit$——————————————————————————$\diamondsuit$

But that "proof" cannot be correct. $\{a^n a^n \mid n > 0\} = \{a^{2n} \mid n > 0\}$. So L_5 is the set of all strings of as whose length is even, and that is a regular language. Where did the proof go wrong?

The incorrect proof states that $a^j a^i$ does not have the form $a^n a^n$. Suppose $j = 4$ and $i = 2$. Then $a^j a^i = a^4 a^2 = a^6 = a^3 a^3$. In fact, as long as $i + j$ is even, $a^j a^i$ *does* have the form $a^n a^n$, where $n = (i + j)/2$.

Can you insist that $i + j$ is odd? Clearly not. The claim is false. You only know that there exist two values i and j where $i < j$ where a^i and a^j take M to the same state, and that is all you can use.

7.6 Be careful not to be sloppy

Having seen a few proofs like the above, all using similar ideas, it is easy to get the idea that it is not necessary to write out all of the details, and instead to skip directly to step 5. But step 4 says what the experiment is; that is, what is the infinite sequence of strings to run M on? If you don't say what the experiment is, you will find yourself making inconsistent statements about that experiment.

There is an easy way to avoid that. Don't skip the details. Write them down and check that what you have written is sensible. Don't expect a person who reads your proof to fill in the details for you. Nonregular languages

8 Programs and Computability

8.1 Programs

With this section, we begin to look at what can be computed by general programs. But what is a general program?

A full definition of a general program is involved and takes us into an area, *automata theory* (ah-TOM-a-tah theory; automata is the plural of automaton (ah-TOM-a-tahn)), that we will not explore in this course for lack of time. So let's settle for a less-than-rigorous defininition of a program.

Program "$\{p(x)$: *body*$\}$" is a program or function called p that performs actions indicated by *body*. In the body, a program says **return** r to indicate that the answer is r. Otherwise, the body is written in *psuedo-code* that you can imagine has been translated into your favorite programming language. We use indentation to show program structure.

Technically the input, or parameter, is always a string. But the input might be an integer, written in base 10. It might have more than one thing encoded in it. For example, input "(25,400)" describes an ordered pair of integers. So we will allow a program with more than one input, as in "$\{q(x,y)$: $\dots\}$".

Program "$\{a(x_1x_2\dots x_n)$: $\dots\}$" takes a parameter string $x =$ "$x_1x_2\dots x_n$"; in the body, x_i refers to the i-th character of x.

Some examples are shown later in this section.

8.1.1 A program is a string

We write a program in quotes because a program is a string. You create a program using a **text-**editor. That point is important for the study of computability. If there are string constants embedded inside the program, I will not write \" for the embedded quotes. There should be no confusion from that.

We refer to program "$\{p(x)$: $\dots\}$" a p. Keep in mind that p is both a program and a string.

8.2 Computability

8.2.1 Computable functions

Definition 8.1. For our purposes, an *algorithm* is a program that stops and produces an answer for every input. It is not allowed to loop forever, and is not allowed to stop without giving an answer.

Definition 8.2. Suppose that Σ and Γ are alphabets and $f : \Sigma^* \to \Gamma^*$ is a function. Program p *computes* function f provided, for every string $s \in \Sigma^*$, when p is run on input s, it eventually stops and returns string $f(s)$. That is, a function is computable if there is an algorithm that computes it.

Definition 8.3. Function f is *computable* if there exists a program that computes f.

8.2.2 Computable decision problems

Definition 8.4. Suppose $A \subseteq \Sigma^*$ is a language over Σ^*. A program p *computes* A provided, for every string $s \in \Sigma^*$, when p is run on input s, it eventually stops and returns 1 if $s \in A$ and returns 0 if $s \notin A$. That is, language A is computable if there is an algorithm that computes the problem of determining whether a given string x is in A.

If p computes A, we also say that p *solves* A and that p *decides* A.

Definition 8.5. If p is a program, define $L(p)$ to be the set of all strings on which program p stops and returns 1. We say that $L(p)$ is the language that p accepts.

Definition 8.6. Language A is *computable* provided there exists a program that computes A. Equivalently, A is computable if there exists a program p that stops on every input and where $L(p) = A$. Computable decision problems are also said to be *decidable*.

Note that computability is not defined in terms of what you or I are clever enough to do. A function or language is computable if *there exists* a program that computes it, regardless of whether any human is or will ever be able to find such a program.

8.2.3 The Church/Turing Thesis

Each programming language is a *model of computation*. Why can we ignore details like which programming language is chosen (within some limits) in the definition of a computable problem? Because every sufficiently general programming language can solve the same problems, as long as you take away restrictions on the amount of memory that the program can use. That observation is captured in the *Church/Turing Thesis*: the class of computable problems is the same for all sufficiently general models of computation.

One hardly needs much to achieve sufficiently general power. A common model of computation is a *Turing machine*, whose memory consists of an infinitely long tape that can store one symbol per cell, and that can only be read and written using a head that can move to the left and right over the tape. One kind of model of computing has only *counters* as memory, where a counter can be incremented, decremented and tested to see whether it is 0. Astoundingly, a machine with only two counters has general power. The input is initially stored in one of the counters, as an integer that represents a string.

8.2.4 The "type" of adjective *computable*

A language can be computable. A function that takes a string and yields a string can be computable. A function that takes a number and yields a number can be computable.

But a program cannot be computable. It makes no sense to talk about a computable program. So please don't ever to that. Make sure that you know what type of thing you have.

8.3 Examples of computable decision problems

It is easy to come up with computable decision problems.

Theorem 8.1. The empty set is computable.

Proof. Recall that language {} is thought of as the following decision problem.

Input. String x
Question. Is $x \in \{\}$?

Of course, the answer to the question is no regardless of what x is, and program "$\{e(x)$: return 0$\}$" computes $\{\}$.

◇————————————————————◇

Theorem 8.2. Language $\{$"b", "abb", "$baba$"$\}$ is computable.

Proof. The following program t computes $\{$"b", "abb", "$baba$"$\}$.

```
"{$t(x)$:
   if $x$ == "$b$"
      return 1
   else if $x$ == "$abb$"
      return 1
   else if $x$ == "$baba$"
      return 1
   else
      return 0
}"
```

◇————————————————————◇

You should be able to use the idea in the proof of Theorem 8.2 to prove the following.

Theorem 8.3. Every finite set is computable.

Theorem 8.4 shows that some nonregular languages are computable.

Theorem 8.4. Language $\{a^n b^n \mid n > 0\}$ is computable.

Proof. Suppose that $\Sigma = \{a, b\}$. To compute $\{a^n b^n \mid n > 0\}$, it suffices to (1) check that there does not occur an a after a b, and (2) count the as, count the bs, and check that the two counts are the same. The following program accomplishes that.

```
"{$p(x_1 x_2 \ldots x_n)$:
   $i = 1$
```

```
    $c_a = 0$
    $c_b = 0$
    while $i \leq n$ and $x_i$ == 'a'
        $i = i + 1$
        $c_a = c_a + 1$
    while $i \leq n$ and $x_i$ == 'b'
        $i = i + 1$
        $c_b = c_b + 1$
    if $i == n + 1$ and $c_a == c_b$
        return 1
    else
        return 0
}"
```

◇———————————————————————◇

Theorem 8.5. Language $\{n \mid n \text{ is a prime integer}\}$ is computable.

Proof. The following program tells whether n is prime.

```
"{$p(n)$:
    if $n < 2$
        return 0;
    $i = 2$
    while $i < n$
        if $n$ mod $i == 0$
            return 0
        $i = i + 1$
    return 1
}"
```

◇———————————————————————◇

8.4 Every regular language is computable

Theorem 8.6. Every regular language is computable.

Proof. Suppose that A is a regular language. That is, there exists a FSM M so that $L(M) = A$. Assume that someone else gives us such a FSM $M = (\Sigma, Q, q_0, F, \delta)$. Here is a program $R(x)$ that solves A. It simply "runs" M on input x.

```
"{$R(x_1x_2 \dots x_n)$:
    $q = q_0$
    $i = 1$
    while $i \leq n$
        $q = \delta(q, x_i)$
        $i = i + 1$
    if $q \in F$
        return 1
    else
        return 0
}"
```

◇————————————————————————◇

8.5 Computable questions about FSMs

A program can take a FSM as an input. It is just a matter of encoding the FSM as a string. Suppose that $M = (\{a, b\}, \{1, 2, 3\}, 1, \{2, 3\}, \delta)$ where the transition function δ is as follows.

δ	a	b
1	1	2
2	3	1
3	1	1

A possible encoding of M as a string is

"{a,b}{1,2,3}1{2,3}(1,a:1)(1,b:2)(2,a:3),(2,b:1),(3,a:1)(3,b:1)".

Obviously, many different encodings would work.

8.5.1 Does M accept x?

Definition 8.7. The *acceptance problem for FSMs* is the following decision problem.

> **Input.** A FSM M (encoded as a string) and a string x.
> **Question.** Does M accept x?

Theorem 8.7. The acceptance problem for FSMs is computable.

Proof. We have shown, above, how to simulate a FSM M on input x. The only difference here is that M is encoded as a string. But that is not a problem; any experienced programmer can write a program that reads the encoding and pulls out all of the features of M.

◇————————————————————◇

8.5.2 Does M accept all strings?

Let's look at a more difficult problem.

Definition 8.8. The *everything problem for FSMs* is the following decision problem.

> **Input.** A FSM M (encoded as a string) with alphabet Σ.
> **Question.** Does M accept all strings in Σ^*.

Solving the everything problem for FSMs might at first seem impossible. After all, there are infinitely many strings, and you can't check them all. But that is an illusion; it is actually quite easy to check whether M accepts all strings.

Theorem 8.8. The everything problem for FSMs is computable.

Proof. Suppose $M = (\Sigma, Q, q_0, F, \delta)$. Some FSMs have states that cannot be reached by any input string. M accepts all strings in Σ^* if every state that can be reached is an accepting state. The hardest part is determining the reachable states, and that is actually easy.

Assume that there is a *mark bit* associated with each state of M that a program can set to 0 or 1. (That is easy to arrange. If M's states are $\{1, \ldots, n\}$, all we need is an array of n boolean values to hold the mark bits.)

```
"{everything($M$):
  // Mark all accessible states
  Set the mark bit of every state to 0.
  Set the mark bit of $q_0$ to 1.
  changed = 1
  while changed == 1
    changed = 0
    for each state $q$ of $M$
      if $q$'s mark bit is 1
        for each symbol $a$ in $\Sigma$
          r = $\delta(q, a)$
          if r's mark bit is 0
            set r's mark bit to 1
            changed = 1
  // Check is there a marked rejecting state
  for each state q of $M$
    if $q$'s mark bit is 1 and $q \notin F$
      return 0
  return 1
}"
```

◇———————————————————————————————————————◇

8.5.3 Does M accept no strings?

Definition 8.9. The *emptiness problem for FSMs* is language $\{M \mid L(M) = \{\}\}$. That is, it is the following decision problem.

> **Input.** FSM M (encoded as a string).
> **Question.** Is it the case that M does not accept any strings?

Theorem 8.9. The emptiness problem for finite state machines is computable.

Proof. The proof is similar to the preceding proof, but the algorithm checks that each reachable state is a rejecting state.

◇———————————————————————————————————————◇

8.5.4 Is $L(M_1) \subseteq L(M_2)$?

Definition 8.10. The *subset problem for FSMs* is the following decision problem.

> **Input.** Two FSMs M_1 and M_2 (encoded as strings).
> **Question.** Is $L(M_1) \subseteq L(M_2)$? That is, is every string in $L(M_1)$ also in $L(M_2)$?

Once again, a shallow thought process leads one to conclude that the subset problem for FSMs is not computable, since there are infinitely many strings to check. A more careful look shows that it is computable.

Theorem 8.10. The subset problem for FSMs is computable.

Proof. We have seen, in Theorems 5.1 and 5.2, that the class regular languages is closed under complementation and intersection. It is important that both theorems are proved by constructive proofs. That is,

1. There is an algorithm that, given a FSM M, produces FSM M' so that $L(M') = \overline{L(M)}$.

2. There is an algorithm that, given FSMs M_1 and M_2, produces FSM M' so that $L(M') = L(M_1) \cap L(M_2)$.

For any two sets A and B,

$$A \subseteq B \leftrightarrow A - B = \{\}.$$

But $A - B = A \cap \overline{B}$. The algorithm first builds FSM M_3 so that $L(M_3) = \overline{L(M_2)}$. Then it builds FSM M_4 so that

$$L(M_4) = L(M_1) \cap L(M_3) = L(M_1) \cap \overline{(L(M_2))} = L(M_1) - L(M_2).$$

So $L(M_1) \subseteq L(M_2) \leftrightarrow L(M_4) = \{\}$. But we have an algorithm (Theorem 8.4) to tell if $L(M_4) = \{\}$.

◇————————————————————◇

8.5.5 Are $L(M_1)$ and $L(M_2)$ the same language?

Definition 8.11. The *equivalence problem for FSMs* is the following decision problem.

> **Input.** Two FSMs M_1 and M_2 (encoded as strings).
> **Question.** Is $L(M_1) = L(M_2)$?

Theorem 8.11. The equivalence problem for FSMs is computable.

Proof. For any two sets A and B, by definition,

$$A = B \leftrightarrow A \subseteq B \wedge B \subseteq A.$$

It suffices to test each of $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$ separately.

◇————————————————————————————————◇

8.6 Computable problems about polynomials

Let's look at problems involving polynomials with integer coefficients, which we simply call polynomials. An input to such a problem might be $5x^2 - 2$ or $x^2 + 1$. A value of x that makes $5x^2 - 2 = 0$ is called a *zero* of polynomial $5x^2 - 2$.

Definition 8.12. The *real-zero problem* takes a polynomial p of variable x as input and asks whether there is a zero of p that belongs to $\mathcal{R}$, the set of real numbers.

For example, polynomial $x^5 - 2x^3 - 16$ has value 0 when $x = 2$, so it is a yes-input to the real-zero problem. Polynomial $4x^2 - 4x + 1$ is also a yes-input, since it has value 0 for $x = 1/2$.

8.6.1 Quadratic single-variable polymomials

A naive person's first thought might be that the zero problem is not computable since an algorithm would have to try every possible number. But it should be clear that the zero problem is computable for quadratic polynomials. The quadratic formula tells you that equation $ax^2 + bx + c = 0$ has a real-valued solution if and only if $b^2 - 4ac \geq 0$.

8.6.2 Arbitrary degree single-variable polymomials

What if polynomials in x are allowed to have any degree? There are formulas for polynomials of degrees up to 4, but there is no formula for polynomials of degree 5 or higher. (The lack of a formula for degree 5 polynomials is one of the celebrated mathematical results of the nineteenth century.) But we don't need a formula, only an algorithm.

There are algorithms for finding zeros of polymonials of arbitrarily high degree. The details are beyond the scope of this class, but you can get a rough idea of how such an algorithm can work. The coefficient with largest absolute value and the polynomial's degree allow you to compute upper and lower bounds on potential zeros. Outside that range, the polynomial is heading toward ∞ or $-\infty$. An algorithm can cut that range up into small pieces and look for an interval where the polynomial changes sign. The polynomial must cross the x-access somewhere in that interval.

Although we have not proved it here, the real-zero problem is solveable for arbitrary polynomials of a single variable.

8.6.3 Multivariate polynomials

A *multivariate polynomial*, such as $xy - y^2 + 9z$, can have any number of different variables. A single-variable polynomial of degree k can have no more than k different zeros. But a multivariate polynomial can have infinitely many zeros. Look at equation $x - y = 0$. Obviously, any pair of values (x, y) is a zero if $x = y$.

The algorithm is very involved, but it turns out that the real-zero problem is computable for arbitrary multivariable polynomials.

Programs and computability

9 Uncomputable Problems

9.1 A problem about polynomials

There is a another problem about multivariate polynomials that is concerned with integer solutions.

Definition 9.1. The *integer-zero problem* takes a multivariate polynomial p as input and asks whether there are integer values (members or $\mathcal{Z}$) for the variables that occur in p that make $p = 0$.

In 1900, mathematician David Hilbert posed a list major challenges in mathematics. The tenth problem in the list was to find an algorithm to solve the integer-zero problem or to show that no such algorithm exists. It was not until 1970 that Hilbert's Tenth Problem was solved, in the negative. Russian mathemetician Yuri Matiyasevich showed that the integer-zero problem is uncomputable.

A proof that Hilbert's Tenth Problem is not computable is far out of reach for us. Matiyasevich relied on work by Martin Davis, Hilary Putnam and Julia Robinson spanning 21 years, and they relied on prior work. But we will be able to prove that some other problems are uncomputable.

9.2 Infinite loops

Recall that we only say that program p computes function f or language L if p stops on every input. But that there are programs that do not stop on every input (that, by definition, do not compute any function or language).

Let's write Run(p, x) to indicate the value that program p returns when it is given input (or parameter) x. Because a program might not always stop, Run(p, x) might not have a value. It is useful to create a special value, $\perp$ (called "bottom"), and say that $\text{Run}(p,\ x) = \perp$ when p runs forever on input x.

You can't know by running p on input x whether it loops forever; it might just take a very, very long time to stop. But, from a mathematical standpoint, p either stops or it doesn't, so either $\text{Run}(p,\ x) = \perp$ or $\text{Run}(p,\ x) \neq \perp$.

When a "value" might be $\bot$, we use relation $\cong$ instead of $=$, where $x \cong y$ is read as "x is equivalent to y."

Definition 9.2. If x and y are strings then $x \cong y \leftrightarrow x = y$. Also, $\bot \cong \bot$. But $x \ncong \bot$ and $\bot \ncong x$ for any string x.

Definition 9.3. *Run*$(p, x){\downarrow}$ (p halts on input x) is equivalent to $\text{Run}(p, x) \ncong \bot$. *Run*$(p, x){\uparrow}$ (p does not halt on input x) is equivalent to $\text{Run}(p, x) \cong \bot$.

9.3 Interpreters

Your familiarity with computers tells you that, except for resource limitations, any computer can run programs written in any programming language. For example, you can run a Python program on a computer by loading a Python interpreter onto it.

Interpreters are important tools of computability theory. An interpreter allows you to take a program (a string) and run it inside some other program. Running a program via an interpreter must produce the same results as running it directly.

Definition 9.4. An *interpreter* is a program I having the property that, for every program p and string x, $\text{Run}(I,\ (p, x)) \cong \text{Run}(p,\ x)$.

It is a crucial property of computability theory that interpreters exist. Proving that is obviously a big chore (you need to write an interpreter) and we will not try to do that.

Theorem 9.1. There exists a program I that is an interpreter.

Because we know that an interpreter exists, it is acceptable to write $\text{Run}(p,\ x)$ within the body of a program, where p is a string that is either a parameter of the program or that is computed by the program. Running p is just a matter to running a fixed program, the interpreter, that can be built into your own program.

9.4 Problems about programs

Some decision problems ask questions about programs. An easy one is: Does program p contain a variable called z? But consider the following decision problem, analogous to the acceptance problem for FSMs.

Definition 9.5. The *acceptance problem for programs* is the following decision problem.

> **Input.** Program p and string x.
>
> **Question.** Is $\text{Run}(p, x) \cong 1$?

An obvious approach to solving the acceptance problem is to run p on input x and see whether the result is 1. But what if p loops forever? Clearly, that approach does not work.

We have seen that the failure of an obvious approach does not allow us to conclude that no algorithm exists. Concluding that the acceptance problem is not computable needs a rock-solid proof. We will give such a proof in a later section.

9.5 An uncomputable decision problem

Now we identify a decision problem that we can prove is uncomputable.

Definition 9.6. The *Halting Problem* is language

$$HLT = \{(p, x) \mid \text{Run}(p, x)\downarrow\}.$$

That is, it is the following decision problem.

> **Input.** Program p and string x.
>
> **Question.** Does p ever stop when it is run on input x?

Theorem 9.2. The Halting Problem is not computable.

Proof.

1. The proof is by contradiction. Start by assuming that the Halting Problem is computable.

Know:	The Halting Problem is computable.
Goal:	**F**.

2. Now we know something that uses term *computable*, and that suggests using a definition. By the definition of a computable decision problem, saying that HLT is computable is equivalent to saying that there exists a program r that stops on all inputs and where, for all y,

$$\mathrm{Run}(r, y) \cong 1 \leftrightarrow y \in \mathrm{HLT},$$

$$\mathrm{Run}(r, y) \cong 0 \leftrightarrow y \notin \mathrm{HLT},$$

But HLT is a set or ordered pairs. It only makes sense to ask if $y \in \mathrm{HLT}$ if y is an ordered pair. So lets say that $y = (p, x)$.

Know:	There exists a program r that halts on all inputs so that, for all p and x, $\mathrm{Run}(r, (p, x)) \cong 1 \leftrightarrow (p, x) \in \mathrm{HLT}$ and $\mathrm{Run}(r, (p, x)) \cong 0 \leftrightarrow (p, x) \notin \mathrm{HLT}$.
Goal:	**F**.

3. When you know there exists something with a particular property, you ask someone else to give you such a thing. Let's do that, and call the program that was given to us r. The fact that r halts on all inputs is implicit in the two equivalences (1) and (2).

Known variables:	r (a program)
Know (1):	For all p and x, $\mathrm{Run}(r, (p, x)) \cong 1 \leftrightarrow (p, x) \in \mathrm{HLT}$.
Know (2):	For all p and x, $\mathrm{Run}(r, (p, x)) \cong 0 \leftrightarrow (p, x) \notin \mathrm{HLT}$.
Goal:	**F**.

4. By the definition of HLT,

$$(p, x) \in \mathrm{HLT} \leftrightarrow \mathrm{Run}(p, x)\downarrow.$$

Known variables:	r (a program)
Know (1):	For all p and x, $\text{Run}(r,(p,x)) \cong 1 \leftrightarrow \text{Run}(p,x)\downarrow$.
Know (2):	For all p and x, $\text{Run}(r,(p,x)) \cong 0 \leftrightarrow \text{Run}(p,x)\uparrow$.
Goal:	**F**.

5. So far everything has been boilerplate for a proof by contradition. We have only used definitions. Now comes the inspiration. Every programmer knows how to write an infinite loop. We will allow ourselves to write "loop forever" in a program to indicate an infinite loop. Let's define program s as follows.

```
"{$s(z)$:
  if Run($r$, $(z, z)$) = 1
    loop forever
  else
    return 1
}"
```

That program looks like it comes out of nowhere, but the discussion after this proof gives motivation for defining it. Program s is written to have two properties.

$$\text{Run}(r,(z,z)) \cong 1 \to \text{Run}(s,z)\uparrow .$$

$$\text{Run}(r,(z,z)) \cong 0 \to \text{Run}(s,z)\downarrow .$$

Both of those properties should be obvious from the definition of s.

Known variables:	r and s (two programs)
Know (1):	For all p and x, $\text{Run}(r,(p,x)) \cong 1 \to \text{Run}(p,x)\downarrow$.
Know (2):	For all p and x, $\text{Run}(r,(p,x)) \cong 0 \to \text{Run}(p,x)\uparrow$.
Know (3):	For all z, $\text{Run}(r,(z,z)) \cong 1 \to \text{Run}(s,z)\uparrow$.
Know (4):	For all z, $\text{Run}(r,(z,z)) \cong 0 \to \text{Run}(s,z)\downarrow$.
Goal:	**F**.

6. Since facts (1) and (2) hold for all p and x, they must hold for $p = s$ and $x = s$. Since facts (3) and (4) hold for all z, they must hold for $z = s$. (Here, we make use of the fact that program s is a string.) Making those substitutions yields the following.

Known variables:	r and s (two programs)
Know (1):	$\text{Run}(r,(s,s)) \cong 1 \leftrightarrow \text{Run}(s,s)\downarrow$.
Know (2):	$\text{Run}(r,(s,s)) \cong 0 \leftrightarrow \text{Run}(s,s)\uparrow$.
Know (3):	$\text{Run}(r,(s,s)) \cong 1 \rightarrow \text{Run}(s,s)\uparrow$.
Know (4):	$\text{Run}(r,(s,s)) \cong 0 \rightarrow \text{Run}(s,s)\downarrow$.
Goal:	**F**.

7. Using known facts (3) and then (2), we get

$$\begin{aligned}\text{Run}(r,(s,s)) \cong 1 &\rightarrow \text{Run}(s,s)\uparrow \\ &\rightarrow \text{Run}(r,(s,s)) \cong 0\end{aligned}$$

If $\text{Run}(r,(s,s)) \cong 1$, that leads to a contradiction. ($\text{Run}(r,\ (s,s))$ cannot be both 1 and 0.) So it is not possible for $\text{Run}(r,(s,s)) \cong 1$.

Using known facts (4) and then (1), we get

$$\begin{aligned}\text{Run}(r,(s,s)) \cong 0 &\rightarrow \text{Run}(s,s)\downarrow \\ &\rightarrow \text{Run}(r,(s,s)) \cong 1\end{aligned}$$

If $\text{Run}(r,(s,s)) \cong 0$, that also leads to a contradiction. So it is not possible for $\text{Run}(r,(s,s)) \cong 0$.

But facts (1) and (2) tell us that $\text{Run}(r,(s,s))$ *must* be either 0 or 1. (After all, either $\text{Run}(s,s)\uparrow$ or $\text{Run}(s,s)\downarrow$.) So no matter what, we have reached a contradiction, and have proved **F**.

◇————————————————————◇

What was the motivation for program s in step 5? Notice that, later in the proof, we are only concerned with what s does when its parameter z is s. But, when z is s, the definition of s look as follows.

```
"{$s(s)$:
   if Run($r$, $(s, s)$) $= 1$
      loop forever
   else
      return 1
}"
```

(That is not really allowed, since we cannot define a function with its parameter being itself, but let's allow it to understand where the definition of s comes from.)

Now remember that $\mathrm{Run}(r, (p, x)) \cong 1$ if and only if $\mathrm{Run}(p, x) \downarrow$ because r was chosen to be a program that solves the halting problem. In the if-statement, s asks r whether s halts on input s. If r says that s halts on input s, then s says, not I don't, and enters an infinite loop. If r says that s does not halt on input s, then s says, yes I do, and s halts and returns 1.

In fact, the proof is quite constructive in the sense that, for every program r that purports to solve the Halting Problem, the proof provides an input (s, s) that r answers incorrectly.

9.6 Diagonalization

The above proof that the Halting Problem is uncomputable uses pairs of strings of the form (s, s). If you think about points in the Cartesian plane, points of the form (x, x) are on the diagonal defined by equation $y = x$. Based on that analogy, the proof that the Halting Problem is uncomputable is called a *proof by diagonalization*. Uncomputable problems

10 Reductions between Problems

A *reduction* is a way of solving one problem, assuming that you already know how to solve another problem.

10.1 Turing reductions

Suppose that A and B are two computational problems. They can be decision problems or functional problems. We need to formalize the idea that, if we already know how to solve B, we can solve A.

Definition 10.1. A *Turing reduction* from A to B is a program that computes A and that is able to ask questions about B at no cost.

Definition 10.2. Say that $A \le_t B$ provided there exists a Turing reduction from A to B.

10.1.1 Examples of Turing reductions

Example 10.1. Suppose that

$$\begin{aligned} A &= \{n \in \mathcal{N} \mid n \text{ is even}\} \\ B &= \{n \in \mathcal{N} \mid n \text{ is odd}\}. \end{aligned}$$

The following program is a Turing reduction from A to B.

```
"{even($x$):
  if $x \in B$ then
    return 0
  else
    return 1
}"
```

Notice that it asks if $x \in B$. That is where it uses B at no cost. Since there exists a Turing reduction from A to B, we know that $A \le_t B$.

Example 10.2. Suppose that

$$\begin{aligned} A &= \{p \mid p \text{ is a quadratic polynomial in } x \text{ and } p \text{ has a real zero}\} \\ B &= \{p \mid p \text{ is a polynomial in } x \text{ and } p \text{ has a real zero}\} \end{aligned}$$

Notice that B allows p to have any degree. If you are allowed to ask about zeros of polynomials of any degree, it is easy to ask questions about quadratic polynomials. The following program is a Turing reduction from A to B, establishing that $A \le_t B$.

```
"{has-zero($p$):
  if $p \in B$ then
    return 1
  else
    return 0
}"
```

Example 10.3. You can do a Turing reduction between two functions. Define $f : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ to $g : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ as follows.

$$\begin{aligned} f(m,n) &= m+n \\ g(m,n) &= m \cdot n \end{aligned}$$

Here is a reduction from g to f. It multiplies by doing repeated additions.

```
"{$g(m,n)$:
  $y = 0$
  for $i = 1, \dots, m$
    $y = f(y,n)$
  return $y$
}"
```

Example 10.4. The preceding three examples showed how to define a Turing reduction between two computable problems. You could just replace the test $x \in B$ or the use of $f(p,n)$ by a program that tells you whether $x \in B$ or that computes $f(p, n)$. But this example shows a Turing reduction between two uncomputable problems. Define

$$\begin{aligned} \text{NOTHLT} &= \{(p, x) \mid \text{Run}(p, x)\uparrow\} \\ \text{HLT} &= \{(p, x) \mid \text{Run}(p, x)\downarrow\} \end{aligned}$$

The following is a Turing reduction from NOTHLT to HLT.

```
"{loops($p, x$):
   if $(p, x) \in HLT$ then
      return 0
   else
      return 1
}"
```

It is important to recognize that the test $(p, x) \in$ HLT is done without the need for a program that carries out that test. It is done for free. That is good because there is no program that solves the halting problem.

10.1.2 Properties of Turing reductions

Suppose that A and B are computational problems. The following theorem should be obvious. Just use the Turing reduction program.

Theorem 10.1. If $A \leq_t B$ and B is computable, then A is computable.

We can turn that around using a tautology that is related to the law of the contrapositive:

$$(P \wedge Q) \rightarrow R \equiv (P \wedge \neg R) \rightarrow \neg Q.$$

Corollary 10.2. If $A \leq_t B$ and A is not computable, then B is not computable.

(A corollary is just a theorem whose proof is obvious or trivial, given a previous theorem.) That suggests a way to prove that a problem B is not computable: choose a problem A that you already know is not computable and show that $A \leq_t B$.

10.1.3 An intuitive understanding of relation $\le_t$

Definition 10.2 defines what $A \le_t B$ means. But it can be helpful to have an intuitive understanding to go along with the definition. Let's look at problems from a viewpoint where languages come in only two levels of difficulty: a computable problem is considered easy and an uncomputable problem is considered difficult. Then, according to Theorem 10.1 and Corollary 10.2, $A \le_t B$ indicates that

(a) A is no harder than B, and

(b) B is at least as hard as A.

If you keep that intuition in mind, you will make fewer mistakes. For example, we have seen that, if A is uncomputable and $A \le_t B$, then B is uncomputable too (since it is at least as difficult as uncomputable problem A). What if A is uncomputable and $B \le_t A$? That only tells you that B is no more difficult than an uncomputable problem. So?

10.2 Mapping reductions

Mapping reductions are a restricted form of reductions that only work for decision problems, but that have some advantages over Turing reductions for decision problems. Suppose that A and B are languages.

Definition 10.3. A *mapping reduction* from A to B is a computable function f such that, for every x, $x \in A \leftrightarrow f(x) \in B$.

Definition 10.4. Say that $A \le_m B$ provided there exists a mapping reduction from A to B.

10.2.1 Examples of mapping reductions

Example 10.5. Suppose that

$$\begin{array}{rcl} A & = & \{n \in \mathcal{N} \mid n \text{ is even}\} \\ B & = & \{n \in \mathcal{N} \mid n \text{ is odd}\} \end{array}$$

$f(x) = x + 1$ is a mapping reduction from A to B. There is no need to write a program (except to observe that $f(x)$ is computable).

Example 10.6. Suppose that

$$\begin{aligned} A &= \{p \mid p \text{ is a quadratic polynomial in } x \text{ and } p \text{ has a real zero}\} \\ B &= \{p \mid p \text{ is a polynomial in } x \text{ and } p \text{ has a real zero}\} \end{aligned}$$

Then $f(x) = x$ is a mapping reduction from A to B.

Example 10.7. Define

$$\begin{aligned} K &= \{p \mid \text{Run}(p, p)\downarrow\} \\ \text{HLT} &= \{(p, x) \mid \text{Run}(p, x)\downarrow\} \end{aligned}$$

Then $f(p) = (p, p)$ is a mapping reduction from K to HLT. Notice that

$$\begin{aligned} p \in K &\leftrightarrow \text{Run}(p, p)\downarrow \\ &\leftrightarrow (p, p) \in \text{HLT}. \end{aligned}$$

showing that the requirement $p \in K \leftrightarrow f(p) \in \text{HLT}$ of a mapping reduction from K to HLT is met.

10.2.2 Properties of mapping reductions

Mapping reductions share some properties with Turing reductions.

Theorem 10.3. If $A \leq_m B$ and B is computable then A is computable.

Proof. Suppose that $A \leq_m B$. That is, there exists a mapping reduction from A to B. Ask someone else to provide a mapping reduction f from A to B. The following program is a Turing reduction from A to B.

```
"{$a(x)$:
   $y = f(x)$
   if $y \in B$
      return 1
   else
      return 0
}"
```

Since $x \in A \leftrightarrow f(x) \in B$, it should be clear that $a(x)$ computes A. So $A \leq_t B$. Now simply use Theorem 10.1.

$\diamondsuit$————————————————————————$\diamondsuit$

Corollary 10.4. If $A \leq_m B$ and A is not computable then B is not computable.

10.3 Using Turing reductions to find mapping reductions

Students often have find it easier to discover Turing reductions than mapping reductions. One way to discover a mapping reduction is to find a Turing reduction first and to convert that to a mapping reduction. You just need to obey two requirements in the Turing reduction from A to B.

(a) The Turing reduction must only ask one question about whether a string y is in B.

(b) The answer that the Turing reduction returns must be the same as the answer (0 or 1) returned by the test $y \in B$.

If you obey those requirements, then you find that your Turing reduction must have the form

```
"{$a(x)$:
  $y = f(x)$
  if $y \in B$
    return 1
  else
    return 0
}"
```

The mapping reduction is f.

We showed earlier that, if A and B are defined by

$$\begin{aligned} A &= \{(p,x) \mid \mathrm{Run}(p,x)\uparrow\} \\ B &= \{(p,x) \mid \mathrm{Run}(p,x)\downarrow\} \end{aligned}$$

then $A \leq_t B$. It is worth noting that $A \not\leq_m B$. The reason is that any Turing reduction from A to B must negate the answer that it gets to the question about membership in B, and that is not allowed in a mapping reduction.

Reductions between problems

11 Using Reductions to Show that Problems are Not Computable

Section 10 provides two tools, Turing reductions and mapping reductions, that we can use to demonstrate that a problem is uncomputable. They are generally much easier to apply than diagonalization. Here are the important facts about reductions from Section 10.

Corollary 10.2. If $A \le_t B$ and A is not computable, then B is not computable.

Corollary 10.4. If $A \le_m B$ and A is not computable then B is not computable.

11.1 Run$(p, x)\uparrow$?

Section 10 defines

$$\begin{aligned} \text{NOTHLT} &= \{(p,x) \mid \text{Run}(p,x)\uparrow\} \\ \text{HLT} &= \{(p,x) \mid \text{Run}(p,x)\downarrow\} \end{aligned}$$

and shows that NOTHLT $\le_t$ HLT. We know from Section 9 that HLT is uncomputable. Relationship NOTHLT $\le_t$ HLT only tells us that NOTHLT is no harder than an uncomputable problem, which tells use nothing about NOTHLT. But it is easy to turn that particular reduction around.

Theorem 11.1. HLT $\le_t$ NOTHLT.

Proof. The following is a Turing reduction from HLT to NOTHLT, establishing that HLT $\le_t$ NOTHLT.

```
"{halts($p, x$):
   If $(p, x) \in$ NOTHLT then
      return 0
   else
      return 1
}"
```

By Corollary 10.2, since HLT is uncomputable, NOTHLT is also uncomputable.

◇————————————————————◇

11.2 The acceptance problem

The acceptance problem for programs is as follows.

$$ACC = \{(p, x) \mid \text{Run}(p, x) \cong 1\}.$$

Theorem 11.2. ACC is uncomputable.

Proof. It suffices to show that HLT $\leq_t$ ACC. Here is a Turing reduction from HLT to ACC. It introduces a new wrinkle: it builds a program on the fly.

```
"{halts($p, x$):
   $r$ = "{$r(z)$: $w$ = Run($p$, $z$); return 1}"
   if $(r, x) \in$ ACC
      return 1
   else
      return 0
}"
```

Clearly

$$\begin{array}{rcll} (r, x) \in ACC & \leftrightarrow & \text{Run}(r, x) \cong 1 & \text{by the definition of ACC} \\ & \leftrightarrow & \text{Run}(p, x)\downarrow & \text{by the definition of } r \\ & \leftrightarrow & (p, x) \in HLT & \end{array}$$

so program halts(p, x) correctly answers the question, is $(p, x) \in$ HLT.

◊————————————————————————————◊

There is really no need for the full power of a Turing reduction here. Function

$$f(p, x) = (\text{"}\{r(z) : w = \text{Run}(p, z); \text{return } 1\}\text{"}, x)$$

is a mapping reduction from HLT to ACC; f is computable and, as we have just shown,

$$(p, x) \in HLT \leftrightarrow f(p, x) \in ACC.$$

11.3 Does p terminate on input 1?

We have seen the trick of creating a program on the fly. With the next reduction, we introduce another trick: make that program ignore its parameter, so that it does the same thing on all strings. Define

$$T_1 = \{r \mid \text{Run}(r, 1)\downarrow\}.$$

That is, instead of asking whether a give program halts on some given string x, T_1 asks whether the program halts on input 1. That might sound easier than the Halting Problem, but it is not.

Theorem 11.3. T_1 is uncomputable.

Proof. It suffices to show that there is a mapping reduction from HLT to T_1; that is, we show that HLT $\leq_m T_1$. The usual way to show that something exists is to produce one; here is a mapping reduction f from HLT to T_1.

$$f(p, x) = \texttt{"}\{r(q) : w = \text{Run}(p, x);\ \text{return}\ \ 1\}\texttt{"}.$$

Certainly, f is computable. All it does is write a program (a string) and return that program. f does not run the program that it builds. Notice that program $r(q)$ runs program p on input x, but ignores the result. Also notice that $r(q)$ ignores q; r does the same thing regardless of the parameter that is passed to it.

Let's refer to program "$\{r(q) : w = \text{Run}(p, x);\ \text{return } 1\}$" as $r_{p,x}$, acknowledging the fact that p and x are built into r, and you cannot write $r_{p,x}$ until you know what p and x are. Notice that

$$\begin{array}{llll}
(p, x) \in \text{HLT} & \to & \text{Run}(p, x)\downarrow & \text{by the definition of HLT} \\
 & \to & \text{Run}(r_{p,x}, q)\downarrow \text{ for every } q & \text{by the definition of } r_{p,x} \\
 & \to & \text{Run}(r_{p,x}, 1)\downarrow & \\
 & \to & r_{p,x} \in T_1 & \text{by the definition of } T_1
\end{array}$$

and

$$\begin{array}{llll}
r_{p,x} \in T_1 & \to & \text{Run}(r_{p,x}, 1)\downarrow & \text{by the definition of } T_1 \\
 & \to & \text{Run}(p, x)\downarrow & \text{by the definition of } r_{p,x} \\
 & \to & (p, x) \in \text{HLT} &
\end{array}$$

Putting those together:

$$(p, x) \in \text{HLT} \leftrightarrow r_{p,x} \in T_1.$$

Since $f(p, x) = r_{p,x}$, that is exactly the requirement for f to be a mapping reduction from HLT to T_1.

$\diamondsuit$————————————————————————————$\diamondsuit$

11.4 Does p terminate on input 2?

Define

$$T_2 = \{r \mid \text{Run}(r, 2)\downarrow\}.$$

It should be obvious how to modify the proof of Theorem 11.3 to show that HLT $\le_m T_2$. But we already know that T_1 is uncomputable, so showing that $T_1 \le_m T_2$ is enough to show that T_2 is uncomputable. Let's do that.

Theorem 11.4. $T_1 \le_m T_2$.

Proof. All we need to do is to transform a question of whether a program a halts on input 1 into an equivalent question of whether another program b_a halts on input 2. That is easy to do: define

$$b_a = \texttt{"}\{b(q) : \text{ return Run}(a, 1)\}\texttt{"}.$$

Clearly,

$$\text{Run}(a, 1)\downarrow \; \leftrightarrow \text{Run}(b_a, 2)\downarrow.$$

That is,

$$a \in T_1 \leftrightarrow b_a \in T_2.$$

So $f(a) = b_a$ is mapping reduction from T_1 to T_2.

$\diamondsuit$————————————————————————————$\diamondsuit$

11.5 The everything problem for programs

Define

$$\text{ALL} = \{p \mid \forall x(\text{Run}(p, x)\downarrow)\}.$$

That is, ALL is the following decision problem.

Input. Program p.

Question. Does p halt on every input?

Theorem 11.5. ALL is uncomputable.

Proof. It certainly is not enough to argue that an algorithm to solve ALL would need to try every input. That is nonsense. Write a program that clearly halts on every input, such as the following.

```
"{stopper($x$)
   return 1
}"
```

Do you need to try it on every input to be sure that it stops on every input? Of course not. Write another program that clearly loops forever on all inputs, such as the following.

```
"{looper($x$)
   while(1)
      do nothing
}"
```

You can see from the structure of the program that it loops forever on all inputs. What we need to show is that there is no program R that takes *any* program p as an input and tells you whether p stops on all inputs.

The proof is a mapping reduction from T_1 to ALL. Define

$$\begin{aligned} r_p &= \text{"}\{r(q):\ \text{return Run}(p,1)\}\text{"} \\ f(p) &= r_p \end{aligned}$$

Since r_p ignores its parameter q, it should be clear from the definition of r_p that

$$\text{Run}(p,1)\downarrow\ \leftrightarrow \forall q(\text{Run}(r_p,q)\downarrow).$$

That is,

$$p \in T_1 \leftrightarrow r_p \in \text{ALL}$$

which means that $f(p) = r_p$ is a mapping reduction from T_1 to ALL.

◇————————————————————————◇

11.6 Complementation and computability

Theorem 11.6. Suppose S is a language over alphabet Σ. If S is a computable then $\overline{S}$ is also computable.

Proof. Suppose that program p computes S. That is, p stops on every input and, for every $x \in \Sigma^*$,

$$\text{Run}(p, x) \cong 1 \leftrightarrow x \in S.$$

The following program computes $\overline{S}$ by flipping answers from 1 to 0 and from 0 to 1.

```
"{complement($x$):
    if Run($p$, $x$) == 1
        return 0
    else
        return 1
}"
```

In fact, it is obvious that Theorem 11.6 extends to an equivalence.

Theorem 11.7. S is computable if an only if $\overline{S}$ is computable.

11.7 Rice's Theorem

Excluding the proof of Theorem 11.1, you should notice similarities in the above proofs. Can we prove a general theorem so that those theorems all become corollaries of the general theorem? Such a theorem would say something like, "It is not computable to determine whether a program has a property that is based solely on what that program does when you run it." But that is much too vague. The first step is to find a precise definition of what that means.

11.7.1 Definitions

Definition 11.1. Programs p and q are *equivalent* if $\text{Run}(p, x) \cong \text{Run}(q, x)$ for every x. We write $p \approx q$ to mean that p and q are equivalent programs.

Suppose that L is a set of programs over alphabet Σ. Define $\overline{L} = \Sigma^* - L$.

Definition 11.2. L *nontrivial* is $L \neq \{\}$ and $L \neq \Sigma*$. That is, neither L nor $\overline{L}$ is empty.

The following theorem is obvious.

Theorem 11.8. *L is nontrivial if and only if $\overline{L}$ is nontrivial.*

Definition 11.3. L *respects equivalence* provided, for every pair of equivalent programs p and q, either p and q are both in L or p and q are both in $\overline{L}$.

The following is immediate from Definition 11.3.

Theorem 11.9. *L respects equivalence if and only if $\overline{L}$ respects equivalence.*

Definition 11.4. Define

$$\text{LOOP} = \text{"}\{\text{LOOP}(x) : \text{loop forever}\}\text{"}$$

to be a program that loops forever on all inputs.

11.7.2 Rice's Theorem

Our goal is to prove Rice's Theorem, stating that every nontrivial set of programs that respects equivalence is uncomputable. We will do that using a lemma and a corollary to the lemma. (A lemma is a theorem that is proved as a step in proving a more important theorem.)

Lemma 11.10. *If L is a nontrivial set of programs that respects equivalence, where $\text{LOOP} \notin L$, then $\text{HLT} \leq_m L$.*

Proof.

1. Suppose that L is a nontrivial set of programs that respects equivalence and where $\text{LOOP} \notin L$.

Known variables:	L
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Goal:	HLT $\leq_m L$.

2. Since L is nontrivial, there must be some program that is a member of L. Ask someone else to provide one. Let's call it Y.

Known variables:	L, Y
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Goal:	HLT $\leq_m L$.

3. For any given p and x, define $r_{p,x}$ as follows.

```
"{$r_{p,x}(z)$:
  $w$ = Run($p$, $x$)
  return $Y(z)$
}"
```

Known variables:	L, Y, $r_{p,x}$
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Goal:	HLT $\leq_m L$.

4. Notice that, for arbitrary p and x,

$$\begin{aligned} (p,x) \in \text{HLT} &\to \forall z(\text{Run}(r_{p,x}, z) \cong \text{Run}(Y, z)) \\ &\to r_{p,x} \approx Y \\ &\to r_{p,x} \in L \qquad \text{since } L \text{ respects equivalence} \end{aligned}$$

Known variables:	L, Y, $r_{p,x}$
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Know (6):	$\forall p \forall x((p,x) \in \text{HLT} \to r_{p,x} \in L)$
Goal:	HLT $\leq_m L$.

5. Also, for arbitrary p and x,

$$\begin{aligned} (p,x) \notin \text{HLT} &\to \forall z(\text{Run}(r_{p,x}, z)\uparrow) \\ &\to r_{p,x} \approx \text{LOOP} \\ &\to r_{p,x} \notin L \qquad \text{since } L \text{ respects equivalence} \end{aligned}$$

Known variables:	L, Y, $r_{p,x}$
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Know (6):	$\forall p \forall x((p,x) \in \text{HLT} \to r_{p,x} \in L)$
Know (7):	$\forall p \forall x((p,x) \notin \text{HLT} \to r_{p,x} \notin L)$
Goal:	HLT $\leq_m L$.

6. Define function

$$f(p,x) = r_{p,x}.$$

Clearly, f is computable, since it only needs to write down program $r_{p,x}$. Putting facts (6) and (7) together,

$$(p, x) \in \text{HLT} \leftrightarrow r_{p,x} \in L.$$

So f is a mapping reduction from HLT to L.

$\diamond$———————————————————————————$\diamond$

Corollary 11.11. If L is a nontrivial set of programs that respects equivalence, where LOOP $\not\in L$, then L is not computable.

Proof. That follows immediately from Lemma 11.10, corollary 10.4and the fact that HLT is uncomputable.

$\diamond$———————————————————————————$\diamond$

Theorem 11.12. (Rice's Theorem) If L is a nontrivial set of programs that respects equivalence, then L is not computable.

Proof. There are two cases: either LOOP $\not\in L$ or LOOP $\in L$.

If LOOP $\not\in L$, then Theorem 11.12 follows immediately from Corollary 11.11.

So consider the case where LOOP $\in$ Ł. Then LOOP $\not\in \overline{L}$. By theorems 11.8 and 11.9, $\overline{L}$ is nontrivial and $\overline{L}$ respects equivalence. So $\overline{L}$ meets the requirements of Corollary 11.11. We conclude that, in this case, $\overline{L}$ is uncomputable. By Theorem 11.7, L is also uncomputable.

$\diamond$———————————————————————————$\diamond$

11.8 Examples of using Rice's theorem

11.8.1 Example: T_1 is uncomputable

Recall that we defined

$$T_1 = \{r \mid \text{Run}(r, 1)\downarrow\}.$$

Let's reprove that T_1 is uncomputable using Rice's Theorem.

Theorem 11.13. T_1 is uncomputable.

Proof. Since some programs halt on input 1 and some don't, T_1 is nontrivial. Suppose that p and q are two equivalent programs. Then

$$\begin{array}{rcll} p \in T_1 & \leftrightarrow & \text{Run}(p,1){\downarrow} & \text{by the definition of } T_1 \\ & \leftrightarrow & \text{Run}(q,1){\downarrow} & \text{since } p \approx q \\ & \leftrightarrow & q \in T_1 & \text{by the definition of } T_1 \end{array}$$

By Rice's Theorem, T_1 is uncomputable.

◇————————————————————————————◇

11.8.2 Example: is $L(p)$ finite?

Define

$$\text{FINITE} = \{p \mid L(p) \text{ is a finite set}\}.$$

FINITE is the following decision problem.

> **Input.** A program p.
>
> **Question.** Is $L(p)$ finite? That is, is $\{x \mid \text{Run}(p,x) \cong 1\}$ a finite set?

Notice that FINITE is not a finite set! For every computable set S, there are infinitely many programs that solve S. (You can make infinitely many variations on a program without changing the set that it decides.) So there are infinitely many programs p where $L(p) = \{\}$, and all of those are members of FINITE.

Theorem 11.14. FINITE is uncomputable.

Proof. FINITE is nontrivial. Some programs answer 1 on only finitely many inputs, and some answer 1 on infinitely many inputs.

Suppose that p and q are two equivalent programs. Then

$$\begin{array}{rcll} p \in \text{FINITE} & \leftrightarrow & L(p) \text{ is a finite set} & \\ & \leftrightarrow & L(q) \text{ is a finite set} & \text{since } p \approx q \\ & \leftrightarrow & q \in FINITE & \end{array}$$

So FINITE respects equivalence. By Rice's Theorem, FINITE is uncomputable.

◇————————————————————————————◇

11.8.3 Example: is $L(p) = \{\}$?

Define

$$\text{EMPTY} = \{p \mid L(p) = \{\}.$$

EMPTY is not an empty set! It is the following decision problem.

> **Input.** A program p.
>
> **Question.** Is it the case that $L(p)\ \ \{\}$? That is, are there no inputs x on which p stops and answers 1?

Theorem 11.15. EMPTY is uncomputable.

Proof. EMPTY is clearly nontrivial. It also respects equivalence.

$$\begin{aligned} p \in \text{EMPTY} &\leftrightarrow L(p) = \{\} \\ &\leftrightarrow L(q) = \{\} && \text{since } p \approx q \\ &\leftrightarrow q \in EMPTY \end{aligned}$$

By Rice's Theorem, EMPTY is uncomputable.

◇————————————————————————————◇

11.9 Are p and q equivalent?

Define

$$\text{EQUIV} = \{(p, q) \mid p \approx q\}.$$

Rice's Theorem has nothing to say about EQUIV because EQUIV is not a set of programs. It is a set of ordered pairs of programs. Nevertheless, we can show that EQUIV is uncomputable.

Theorem 11.16. EQUIV is uncomputable.

Proof. Define

$$\text{NEVERHALT} = \{p \mid \forall x(\text{Run}(p, x)\uparrow).$$

Rice's theorem does tell us that NEVERHALT is uncomputable. Notice that

$$NEVERHALT = \{p \mid p \approx \text{LOOP}\}.$$

Function f defined by

$$f(p) = (p, \text{LOOP})$$

is a mapping reduction from NEVERHALT to EQUIV, since

$$\begin{aligned} p \in NEVERHALT &\leftrightarrow p \approx \text{LOOP} \\ &\leftrightarrow (p, \text{LOOP}) \in \text{EQUIV} \end{aligned}$$

11.10 K

Define

$$K = \{p \mid \text{Run}(p, p) \not\equiv \bot\}.$$

K is a set of programs, but it does not respect equivalence. Let's try to show that K respects equivalence to see where the proof breaks down.

$$\begin{aligned} p \in K &\leftrightarrow \text{Run}(p, p)\downarrow \\ &\leftrightarrow \text{Run}(q, p)\downarrow \quad \text{since } p \approx q \end{aligned}$$

But what q does on input p is irrelevant to determining whether $q \in K$. All that matters is what q does on input q.

Nevertheless, we can show:

Theorem 11.17. K is uncomputable.

Proof. Rice's theorem is not a help here. But it suffices to show that HLT $\leq_m$ K. For arbitrary p and x, define $r_{p,x}$ as follows.

```
= "{$r_{p,x}(z)$:
   $w$ = Run($p$, $x$)
   return 1
}"
```

Notice that $r_{p,x}$ ignores its parameter, z. It is evident that

$$\begin{aligned} (p, x) \in HLT &\rightarrow \text{Run}(p, x)\downarrow \\ &\rightarrow \forall z(\text{Run}(r_{p,x}, z)\downarrow) \\ &\rightarrow \text{Run}(r_{p,x}, r_{p,x})\downarrow) \\ &\rightarrow r_{p,x} \in K \end{aligned}$$

and

$$\begin{aligned}(p,x) \notin HLT &\rightarrow \text{Run}(p,x)\uparrow \\ &\rightarrow \forall z(\text{Run}(r_{p,x}, z)\uparrow) \\ &\rightarrow \text{Run}(r_{p,x}, r_{p,x})\uparrow) \\ &\rightarrow r_{p,x} \notin K\end{aligned}$$

which tells us that

$$f(p,x) = r_{p,x}$$

is a mapping reduction from HLT to K.

11.11 Concrete examples

Without concrete examples, it can be easy to believe that our theorems about problems being uncomputable are only of abstract, mathematical significance, and have no bearing on the real world. So let's look at some concrete examples to see that the real world is not immune to mathematical theorems.

11.11.1 The 3n+1 problem

The $3n+1$ problem concerns an infinite collection of sequences of integers. Select a positive integer n to start a sequence. Follow it by $n/2$ if n is even and by $3n+1$ if n is odd. Stop the sequence at 1. The sequence starting with 9 is (9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

It is not obvious that the $3n+1$ sequence stops for all starting values. It is conceivable that it gets into a cycle. It is also conceivable that, for some starting values, the numbers in the $3n+1$ sequence keep getting larger and larger, without bound. In fact, nobody knows whether every $3n+1$ sequence is finitely long. But we can always make a conjecture.

Conjecture 11.1. The $3n+1$ sequence is finitely long for every start value.

But look at the following program.

```
"{test($n$)
  $i = n$
  while $i > 1$
    if $i$ is even
      $i = i/2$
    else
      $i = 3i + 1$
}"
```

Can you tell whether test(n) is in ALL? If it is, then Conjecture 11.1 is true. If not, then Conjecture 11.1 is false.

11.11.2 Does program p test whether a number is prime?

Now suppose that you are serving as a grader for a computer programming course. One of the assignments for that course asks students to write a program that reads an input $n > 1$ and tells whether n is prime. As grader, you are tasked with determining whether each submission is correct, with the sole criterion for correctness being that the program correctly determines whether n is prime for every integer n. (In the programming language being used, integers can be arbitrarily large, so you can't try the program on a finite range of integers to decide whether it works.)

To make sure that you are ready, you write your own program p to tell if a number is prime. Now, given a student submission q, the problem is to determine whether $q \approx p$. But that is uncomputable! Could that possibly be a problem? Suppose that a particularly devious student submits the following program.

```
"{$q(n)$
  $i = n$
  while $i > 1$
    if $i$ is even
      $i = i/2$
    else
      $i = 3i + 1$
  $i = 2$
  while $i < n$
```

```
        if $n$ mod $i$ == 0
            return 0
        i = i + 1
    return 1
}"
```

You notice that, if Conjecture 11.1 is true, the submitted program q is correct. But if Conjecture 11.1 is false, then there are values n on which q loops forever, meaning that q is incorrect. In order to grade q according to the grading criterion, you must determine whether Conjecture 11.1 is true!

11.11.3 Goldbach's conjecture

The following conjecture is due to Goldbach.

Conjecture 11.2 Every even integer that is greater than 2 is the sum of two prime integers.

For example, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 5 + 5$, etc. Nobody knows whether Goldbach's conjecture is true, and it appears to be a very difficult nut to crack. But we can write the following program, which contains an infinite loop that checks, for each even number n, whether there are two prime numbers whose sum is n. If it finds an even number n that is not the sum of two prime numbers, it stops. Otherwise, it loops forever.

```
"{goldbach()
    $n = 4$
    while 1
        $i = 2$
        found = 0
        while not found and $i < n$
            if $i$ is prime and $n - i$ is prime
                found = 1
            $i = i + 1$
        if not found
            return 0
        $n = n + 2$
}"
```

To answer Goldbach's conjecture, all you need to do is ask whether program goldbach ever stops. You can ask whether it is in ALL or in T_1 or in a variety of languages because goldbach ignores its input. Using reductions to show problems are not computable

12 Computational Complexity and Polynomial Time

A program is only said to compute, or decide, a problem if it eventually stops on every input. For computability, it does not matter how long the program takes to produce an answer.

But, from a practical standpoint, time matters. The data switches that form the backbone of the internet process a data packet roughly every 10-20 nanoseconds. They can only afford to run extremely fast algorithms. For most everyday purposes, an algorithm that gets its answer in a few seconds is fast enough, and for some problems, a program that takes a few minutes or hours is fast enough.

With this section we start to look at what can be computed efficiently. To do that, we need to choose a reasonable definition of an "efficient" algorithm, and to study what problems can be solved efficiently under that definition.

A definition of efficiency cannot be based on any fixed amount of time. The larger the input is, the longer we expect a program to take, so a reasonable notion of efficiency should be concerned with the time that a program takes as a function of the length of the input. Also, a faster computer will produce an answer faster even for the same algorithm, and we need a way to get that speed out the way.

12.1 The class P

A program performs a sequence of instructions, and the time that it uses is just a count of the number of instructions that it performs before it stops. In this view, time has no units; it is a pure number. We assume that it takes at least one instruction to look at a symbol in the input and at least one instruction to write one symbol in the output.

Definition 12.1. *Time(p,x)* is the number of instructions that program p takes when it is run on input x. If $\text{Run}(p, x) \cong \bot$, then $\text{Time}(p,x) = \infty$.

Definition 12.2. Let $f : \mathcal{N} \to \mathcal{N}$. A program p runs in *time $O(f(n))$* if there exists constants a and c so that, for all $n > a$ and all strings x of length n, $\text{Time}(p, x) \leq c \cdot f(n)$.

Definition 12.3. Program p runs in *polynomial time* if there exists a positive integer k so that p runs in time $O(n^k)$. When p runs in polynomial time, we say that p is a *polynomial-time algorithm*.

Definition 12.4. *P* is the class of all *decision problems* that have polynomial-time algorithms.

12.2 Examples of problems that are in P

12.2.1 Example: is x a palindrome?

A *palindrome* is a string such as "*aabaa*" that is the same forwards and backwards. The *Palindrome Problem* is the following decision problem.

> **Input.** String x.
>
> **Question.** Is x a palindrome?

The Palindrome Problem is in P. You should be able to figure out an algorithm that solves the Palindrome Problem in time $O(n)$.

12.2.2 Example: does FSM M accept x?

We have seen that it is computable to determine if a given FSM M accepts a given string x. It should also be clear that an algorithm can do that in time that is proportional to the product of the length of the description of M and the length of x. (You should be able to do much better than that, but it is fast enough.) If the input has length n, that is surely $O(n^2)$ time.

12.2.3 Example: is $x \cdot y = z$?

How might you solve the following decision problem?

> **Input.** Three positive integers x, y and z.
>
> **Question.** Is $x \cdot y = z$?

The obvious thing to do is to multiply x and y and check whether the answer is z. It is important to notice that the time to do that is not a constant, since x, y and z can be very large. When a number occurs in the input, its length is the number of digits needed to write it down. For example, 490 has length 3. The algorithm that you learned in elementary school multiplies an i-digit number by a j-digit number in time $O(ij)$. The length of the input is the total number of symbols needed. Clearly, if x has length i and y has length j and the total length of the input if n, then $i < n$ and $j < n$, and it is possible to check an integer product in time $O(n^2)$. That is polynomial time.

12.2.4 Example: is x a prime number?

The *Primality Problem* is the following decision problem.

> **Input.** A positive integer x.
>
> **Question.** Is x prime?

Here is an algorithm that solves the Primality Problem.

```
"{prime($x$):
  $i = 2$
  while $i < x$
    if $n$ mod $i == 0$
      return 0
    $i = i + 1$
  return 1
}"
```

How much time does that algorithm take? It goes around the loop $x - 2$ times. If x is n digits long, then x is in the rough vicinity of 10^n. (Assuming no leading 0s, $10^{n-1} \le x < 10^n$.) The division algorithm that you learned in elementary school divides an m-digit number by an n-digit number in time $O(mn)$. Puting that all together, we find that our algorithm for testing whether an integer is prime takes time $O(n^2 10^n)$. But function $f(n) = 10^n$ grows faster than any polynomial. That is, for every k,

$$\lim_{n\to\infty} \frac{10^n}{n^k} = \infty.$$

Our primality-testing algorithm is not a polynomial-time algorithm.

What does that tell us about whether the Primality Problem is in P? Absolutely nothing! You can write a bad algorithm for any computable problem. The issue is not whether there is a bad algorithm to solve the Primality Problem, but whether there is a polynomial-time algorithm for it.

As it turns out, the primality problem is in P. It was long conjectured to be in P, and was shown to be in P in 2003.

12.3 The Validity Problem for Propositional Logic

Chapter 1 defines the notion of a valid propositional formula. The *Validity Problem for Propositional Logic* (or simply the Validity Problem) is the following decision problem.

> **Input.** A propositional formula ϕ.
>
> **Question.** Is ϕ valid?

You already know an algorithm that solves that problem: truth tables. Suppose that ϕ has v variables. Then a truth table for ϕ has 2^v rows, and determining validity takes time at least 2^v.

Determining whether an algorithm runs in polynomial time requires determining the algorithm's running time in terms of the length n of the input, and v is surely shorter than the total length of ϕ. But as long as we allow long variable names (such as x_1, x_2, ...), it is easy to write an interesting propositional formula of length n with at least $\sqrt{n}$ variables. A little calculus shows that

$$\lim_{n\to\infty} \frac{2^{\sqrt{n}}}{n^k} = \infty$$

for every k, so the truth table algorithm does not run in polynomial time.

What does that have to say about whether the Validity Problem is in P? Absolutely nothing! Nobody knows a polynomial time algorithm for the Validity Problem, but that lack of knowledge also is not convincing evidence that the Validity Problem is not in P.

Beginning with the next section, we begin to address the question of whether the Validity Problem is in P. Computational complexity and polynomial time

13 Nondeterminism and NP

13.1 Mersenne's conjecture

In 1644, Marin Mersenne made what came to be known as Mersenne's conjecture: $2^n - 1$ is prime for n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127 and 257. and is composite for all other positive integers $n \leq 257$.

Mersenne's conjecture stood until 1903 when Frank Cole made a presentation that put it to rest. The presentation was quite short. By starting with 2, successively doubling and finally subtracting 1, Cole showed that

$$2^{67} - 1 = 147,573,952,589,676,412,927.$$

Then he wrote down two numbers and multiplied them together.

$$\begin{array}{lr} & 761{,}838{,}257{,}287 \\ \times & 193{,}707{,}721 \\ \hline & 147{,}573{,}952{,}589{,}676{,}412{,}927 \end{array}$$

That was all it took to convince Cole's audience that Mersenne's conjecture was mistaken. (Other mistakes in it were discovered later.)

But where did Cole get the factors of $2^{67} - 1$? He said that it took "three years of sundays" to find them.

In idealized form, our system of justice is supposed to work as follows. First, the police gather evidence. Then, the prosecutor presents the case to a jury. Finally, the jury rules on whether the evidence is convincing. If the jury does not find the evidence convincing, the jurors are not required to go out and find new evidence. The case is over.

Frank Cole played the role of police and prosecutor and his audience played the role of jury. It can take much less time to present a case than it does to find the evidence.

13.2 Evidence checkers

We can break down testing whether string x is in language A into two parts: finding evidence and checking the evidence.

Definition 13.1. A *polynomial-time evidence checker* for language A is a program check(e, x) where there exists a positive integer k so that

1. check(e, x) runs in polynomal time (in the length of ordered pair (e, x)).
2. If $x \in A$ then there is a string e (the evidence) where $|e| \leq |x|^k$ and Run(check, $(e, x)) \cong 1$. That is, members of A have short, easy to check evidence that they are members of A.
3. If $x \notin A$ then there does not exist any string e so that Run(check, $(e, x))$ $\cong 1$. That is, the evidence checker cannot be fooled into believing that $x \in A$ when in fact $x \notin A$.

There is an important asymmetry in evidence checkers. If $x \in A$, then there must be checkable evidence that $x \in A$. But if $x \notin A$, no evidence is required that $x \notin A$.

13.3 Definition of NP

Definition 13.2. *NP* is the class of all decision problems that have polynomial-time evidence checkers.

For example, an integer x is *composite* if $x > 1$ and x is not prime. The smallest composite number is 4 $(= 2 \cdot 2)$. Define

$$\text{COMPOSITE} = \{x \in \mathcal{N} \mid x \text{ is composite}\}.$$

It is easy to see that COMPOSITE is in NP. (Frank Cole showed how.) The following is a polynomial-time evidence checker for COMPOSITE where the evidence e should be a factor of x and the checker verifies that.

```
"{composite($e, x$):
  If $1 < e < x$ and $x$ mod $e == 0$
    return 1
  else
    return 0
}"
```

A simpler way to present an evidence checker is to list (1) the input, (2) the evidence and (3) the conditions that needs to be satisfied for the evidence to be convincing.

Evidence checker for COMPOSITE	
Input	Positive integer x
Evidence	Positive integer e
Requirement	$1 < e < x$ and $x \bmod e = 0$

13.4 Examples of problems in NP

13.4.1 Is a given propositional formula satisfiable?

Definition 13.3. A propositional formula ϕ is *satisfiable* if there exists a truth-value assignment for the variables in ϕ that makes ϕ true.

Definition 13.4. The *Satisfiability Problem for Propositional Logic* (SATPL) is the following decision problem.

Input. A propositional formula ϕ.
Question. Is ϕ satisfiable?

Theorem 13.1. SATPL is in NP.

Proof. All we need is a polynomial-time evidence checker for SATPL. If you think about a truth-table for ϕ, you only need to look at one row to determine that ϕ is satisfiable.

Evidence checker for SATPL	
Input.	Propositional formula ϕ
Evidence.	Truth value assignment a
Requirement.	$(a \dashv \phi)$ is true.

13.4.2 Does a graph have a small vertex cover?

Definition 13.5. Suppose that $G = (V, E)$ is a simple graph. A *vertex cover* of G is a subset $C \subseteq V$ such that, for every edge $\{u, v\} \in E$, $C \cap \{u, v\} \neq \{\}$. That is, every edge must be incident on at least one member of the vertex cover C.

Definition 13.6. The *Vertex Cover Problem* (VCP) is the following decision problem.

Input. A simple graph G and a positive integer k.
Question. Does there exist a vertex cover C of G where $|C| \le k$?

Theorem 13.2. VCP $\in$ NP.

Proof. Decision problems in NP are often stated as a question about whether something exists. For example, VCP asks whether a vertex cover of a limited size exists. To find a polynomial-time evidence checker, use the thing whose existence is questioned as the evidence. The following is an evidence checker for VCP.

Evidence checker for VCP	
Input	Simple graph $G = (V, E)$ and positive integer k
Evidence	Set $C \subset V$
Requirement	$\lvert C\rvert \le k$ and C is a vertex cover of G.

◇———◇

13.4.3 Can a list of integers be partitioned equally?

Definition 13.7. A list of positive integers x_1, x_2, $\ldots$, x_n is *equally partitionable* if there exists an index set $I \subseteq \{1, 2, \ldots n\}$ such that

$$\sum_{i \in I} x_i = \sum_{i \notin I} x_i.$$

For example, suppose the list of integers is $x_1 = 14$, $x_2 = 10$, $x_3 = 5$, $x_4 = 7$, $x_5 = 2$, $x_6 = 4$, $x_7 = 6$. Index set $\{1,\ 6,\ 7\}$ equally partitions that list since $x_1 + x_6 + x_7 = 14 + 4 + 6 = 24$ and $x_2 + x_3 + x_4 + x_5 = 10 + 5 + 7 + 2 = 24$.

Definition 13.8. The Partition Problem (PP) is the following decision problem.

> **Input.** A list x_1, x_2, …, x_n of positive integers.
> **Question.** Is x_1, x_2, …, x_n equally partitionable?

Theorem 13.3. PP $\in$ NP.

Proof. List x_1, x_2, …, x_n is equally partitionable *there exists* an index set I so that

$$\sum_{i \in I} x_i = \sum_{i \notin I} x_i.$$

That suggests using I as the evidence.

Evidence checker for PP	
Input	List x_1, x_2, …, x_n of positive integers
Evidence	Index set $I \subseteq \{1, \ldots n\}$
Requirement	$\sum_{i \in I} x_i = \sum_{i \notin I} x_i$

◇————————————————————◇

13.5 Every problem in NP is computable

Theorem 13.4. If $A \in$ NP then A is computable.

Proof. Suppose that $A \in$ NP. Let $c(e, x)$ be a polynomial-time evidence checker for A. By the definition of a polynomial-time evidence checker, there is an integer k so that $x \in A$ if and only if there exists a string e with $|e| \le |x|^k$ and $c(e, x) \cong 1$.

An algorithm can decide whether $x \in A$ by computing $c(e, x)$ for every string e where $|e| \le x^k$, answering yes if any of those yields 1.

◇————————————————————◇

13.6 Every problem in P is also in NP

Theorem 13.5. Every language that is in P is also in NP.

Proof. Suppose that A is a language in P. By definition, that means there is a polynomial-time algorithm inA(x) where Run(inA, x) $\cong 1 \leftrightarrow x \in A$. The following is a polynomial-time evidence checker for A. It does not need the evidence, so it ignores the evidence.

Evidence checker for A	
Input	x
Evidence	any string e
Requirement	Run(inA, x) $\cong 1$

$\diamondsuit$——$\diamondsuit$

13.7 The P = NP question

Think of a problem in NP as a kind of puzzle. Solving a puzzle requires finding a solution, which amounts to evidence that the puzzle has a solution. Often, the solution for a puzzle in the newspaper or a book is provided, and peeking at the solution is much less time consuming (though less satisfying) than finding the solution yourself.

Theorem 13.5tells us that P $\subseteq$ NP. But is NP $\subseteq$ P? If NP $\subseteq$ P, then, at least up to a polynomial, peeking at the solution is not helpful; you can just find the solution yourself.

If $NP \subseteq$ P then P = NP. Surprisingly, nobody knows whether P = NP. It is widely *conjectured* that P $\neq$ NP, but we have already seen that a conjecture can stand for over 200 years only to be overturned. Nondeterminism and NP

14 NP-Completeness

14.1 "Easy" and "hard" problems

In Section 10, we only considered two levels of difficulty of problems: computable problems and uncomputable problems. Starting with Section 12, we have begun by considering only just two levels of difficulty, where we think of a problem as "easy" if it is in P and as "hard" if it is not in P. Let's refer to the class of decision problems that are not in P (the "hard" problems) as $\overline{\text{P}}$.

The previous section introduced a third level of difficulty, NP. A problem that is in P is also in NP and every problem that is in NP is computable.

A common misconception is that NP $= \overline{\text{P}}$. That is not the case at all. Can you think of a problem that is in $\overline{\text{P}}$ but not in NP? How about the Halting Problem (HLT)? Since every problem in NP is computable (Theorem 13.4), HLT is not in NP. HLT is also not in P, so it is in $\overline{\text{P}}$.

Our ultimate goal is to find problems that are in NP but not in P. That is, they are not in P, but are only slightly outside of P since they are in NP. This section identifies "hardest" problems in NP, which are the best candidates for languages that are in NP but not in P. In overview:

1. We define polynomial-time mapping reductions and relation $A \le_p B$ where, if $A \le_p B$ and $B \in \text{P}$ then $A \in \text{P}$. Intuitively, you can think of $A \le_p B$ as saying that B is at least as hard as A (in our new two levels of difficulty P and $\overline{\text{P}}$).

2. We define a problem to be NP-complete if it is among the hardest problems in NP. That is, it must be in NP and it must be at least as hard as every other problem in NP.

3. Section 15 identifies some NP-complete problems. In this section, we look at the consequences of a problem being NP-complete.

14.2 Polynomial-time mapping reductions

Definition 14.1. Suppose that A and B are languages (decision problems). A *polynomial-time mapping reduction from A to B* is a function f where

(a) f is computable in polynomial time.

(b) For every string x, $x \in A \leftrightarrow f(x) \in B$.

The only difference between a polynomial-time mapping reduction and the mapping reductions defined in Section 10 is the requirement that f must not merely be computable, but must be computable in polynomial time. It should come as no surprise that polynomial-time mapping reductions have properties that are similar to unrestricted mapping reductions, but with respect to P and $\overline{\text{P}}$ rather than with respect to computable and uncomputable problems.

Theorem 14.1. If $A \leq_p B$ and $B \in \text{P}$ then $A \in \text{P}$.

Proof.

1. Ask someone else to choose arbitrary decision problems A and B, and suppose that $A \leq_p B$ and $B \in \text{P}$.

Known variables:	A, B
Know (1):	$A \leq_p B$.
Know (2):	$B \in \text{P}$.
Goal:	$A \in \text{P}$.

2. Since $A \leq_p B$, there exists a polynomial-time mapping reduction from A to B. Ask someone else to provide one, and call it f.

Known variables:	A, B, f
Know (1):	$A \leq_p B$.
Know (2):	$B \in \text{P}$.
Know (3):	f is a polynomial-time mapping reduction from A to B.
Goal:	$A \in \text{P}$.

3. By the definition of a polynomial-time mapping reduction, we know two things.

(a) $f(x)$ is computable in polynomial time. That means there exists a polynomial-time algorithm $F(x)$ that computes $f(x)$. But a polynomial-time algorithm is required to run in time $O(n^k)$ for some particular positive integer k, where n is the length of the input. Since $F(x)$ and k exist, let's get them from someone else.

(b) $x \in A \leftrightarrow f(x) \in B$ for every x.

Known variables:	A, B, f, F, k
Know (1):	$A \leq_p B$.
Know (2):	$B \in$ P.
Know (3):	f is a polynomial-time mapping reduction from A to B.
Know (4):	Algorithm $F(x)$ computes $f(x)$ in time $O(n^k)$ where $n = \|x\|$.
Know (5):	$\forall x(x \in A \leftrightarrow f(x) \in B)$
Goal:	$A \in$ P.

4. Since $B \in P$, there must exist a polynomial-time algorithm $b(y)$ that tells whether $y \in B$. By the definition of a polynomial-time algorithm, $b(y)$ takes time $O(m^j)$ for a particular positive integer j, where $m = |y|$. Ask someone else to provide algorithm $b(y)$ and integer j.

Known variables:	A, B, f, F, k, b, j
Know (1):	$A \leq_p B$.
Know (2):	$B \in$ P.
Know (3):	f is a polynomial-time mapping reduction from A to B.
Know (4):	Algorithm $F(x)$ computes $f(x)$ in time $O(n^k)$ where $n = \|x\|$.
Know (5):	$\forall x(x \in A \leftrightarrow f(x) \in B)$
Know (6):	Algorithm $b(y)$ tells whether $y \in B$ in time $O(m^j)$ where $m = \|y\|$.
Goal:	$A \in$ P.

5. Consider the following program.

```
"{$a(x)$:
   $y = F(x)$
   if $b(y) == 1$
      return 1
   else
      return 0
}"
```

It is clear that $a(x)$ correctly answers the question "is $x \in A$," since

$$\begin{array}{rcll} a(x) = 1 & \leftrightarrow & b(F(x)) = 1 & \text{by inspection of } a \\ & \leftrightarrow & b(f(x)) = 1 & \text{by fact (4)} \\ & \leftrightarrow & f(x) \in B & \text{by fact (6)} \\ & \leftrightarrow & x \in A & \text{by fact (5)} \end{array}$$

We can also show that $a(x)$ runs in polynomial time. To see that, suppose that $|x| = n$. Computing $y = F(x)$ takes time at most $c_1 n^k$ for some constant c_1. But in t steps, F cannot write down a string that is more than t symbols long. So $m = |y| \le c_1 n^k$. Running $b(y)$ takes $c_2 m^j \le c_2(c_1 n^k)^j = c_3 n^{jk}$ steps, where $c_3 = c_2 c_1^j$. The total time is $O(n^{jk})$.

So $A \in \text{P}$ since $a(x)$ is a polynomial-time algorithm that solves A.

◇―――――――――――――――――――――――――――――――――◇

Corollary 14.2. If $A \le_p B$ and $A \notin \text{P}$ then $B \notin \text{P}$.

Proof. Use Theorem 14.1 and tautology

$$(P \wedge Q) \rightarrow R \;\equiv\; (P \wedge \neg R) \rightarrow \neg Q.$$

◇―――――――――――――――――――――――――――――――――◇

Theorem 14.3. If $A \le_p B$ and $B \le_p C$ then $A \le_p C$. That is, relation $\le_p$ is *transitive*.

Proof. Suppose that $f(x)$ is a polynomial-time mapping reduction from A to B and $g(y)$ is a polynomial-time mapping reduction from B to C. By the definition of a mapping reduction, for every x and y,

$$\begin{aligned} x \in A &\leftrightarrow f(x) \in B && (1) \\ y \in B &\leftrightarrow g(y) \in C && (2) \end{aligned}$$

Define $h(x) = g(f(x))$. Notice that

$$\begin{aligned} x \in A &\leftrightarrow f(x) \in B && \text{by (1)} \\ &\leftrightarrow g(f(x)) \in C && \text{by (2)} \\ &\leftrightarrow h(x) \in C && \text{by the definition of } h(x) \end{aligned}$$

Also, $h(x)$ can be computed in polynomial time. If $f(x)$ is computable in time $O(n^k)$ and $g(y)$ is computable in time $O(n^j)$ then $h(x) = g(f(x))$ can be computed in time $O(n^{jk})$ by an argument similar to the one in step 5 of the proof of Theorem 14.1.

14.3 Definition of an NP-complete problem

Suppose that you want to find the tallest person t in a room. The first requirement, of course, is that person t must be in the room. The second is that person t must be at least as tall as every other person in the room.

Similarly, a decision problem A is a hardest problem in NP if A is in NP and A is at least as hard as every other problem in NP. A hardest problem in NP is called an NP-complete problem.

Definition 14.2. Suppose that A is a language. Say that A is *NP-complete* if

(a) $A \in NP$

(b) For every language $X \in NP$, $X \leq_p A$.

Since A is in NP, the second condition says that A is as hard as *every* problem in NP, including A itself. That is okay: $A \leq_p A$ is clearly true. A is at least as hard as itself.

14.4 Consequences of NP-completeness

It is not obvious that there exists an NP-complete problem. In Section 15, we will see some problems that are provably NP-complete. But right now, let's ask what NP-completeness tells us about a problem.

Our goal is to identify problems that are in NP but not in P. But nobody knows whether there exist *any* problems that are in NP that are not in P! Clearly, NP-completeness does not take us to our goal.

But suppose, for the sake of argument, that it turns out that P $\neq$ NP, and there is at least one language D in NP $-$ P. Also, suppose that problem E is NP-complete. Since $D \in$ NP, it must be the case that $D \leq_p E$. (*All* languages in NP polynomial-time reduce to an NP-complete problem.) Since $D \notin$ P, by Corollary 14.2, $E \notin P$. We have just proved the following.

Theorem 14.4. If P $\neq$ NP and E is NP-complete then $E \notin P$.

On the other hand, what if P = NP? By definition, an NP-complete problem is in NP, so if P = NP, then an NP-complete problem is also in P. That does not mean the problem is not NP-complete. It just means that NP-completeness is not interesting.

It is widely conjectured that P $\neq$ NP. But nobody knows if the conjecture is true.

Conjecture 14.1 P $\neq$ NP.

What would happen if someone finds a polynomial-time algorithm for an NP-complete problem? The following theorem tells you.

Theorem 14.5. If E is NP-complete and $E \in$ P then P = NP.

Proof. Suppose X is an arbitary problem in NP. Since E is NP-complete, $X \leq_p E$. By Theorem 14.1, since X polynomial-time reduces to a problem that is in P, X is also in P.

So every problem in NP is also in P. That is NP $\subseteq$ P. Since P $\subseteq$ NP (Theorem 13.5), P = NP.

◇————————————————————————◇

So, if you are so inclined, you know how to prove that Conjecture 14.1 is wrong. Just find a polynomial-time algorithm for a problem that is known to be NP-complete. But be careful. Every year, a few people have tried to do exactly that. But their algorithms either do not run in polynomial time or do not work.

NP-completeness

15 Examples of NP-Complete Problems

15.1 SAT

Section 13.4 defines the Satisfiability Problem for Propositional Logic (SATPL). We have seen that SATPL is in NP, and noticed in (Section 12) that appears to be difficult to solve.

Here, we look at a restriction of that problem to a certain kind of propositional formula.

Definition 15.1. A *literal* is either a variable or its negation. In this section, we will use lower case variables such as x, y and z as propositional variables. Rather than using $\neg y$ to indicate negated variable y, we write $\overline{y}$. Literal x is a *positive literal* and $\overline{y}$ is a *negative literal*.

Definition 15.2. A *clause* is a disjunction ($\vee$) of one or more literals. For example, $x \vee \overline{z} \vee y$ is a clause. A literal by itself is a clause with just one literal.

Definition 15.3. A *clausal formula* is a conjunction ($\wedge$) of one or more clauses. For example, $(x) \wedge (\overline{y} \vee z) \wedge (\overline{x} \vee y \vee \overline{z})$ is a clausal formula.

Definition 15.4. *SAT* is the following decision problem.

> **Input.** A clausal propositional formula ϕ.
> **Question.** Is ϕ satisfiable?

Theorem 15.1. SAT $\in$ NP.

Proof. Theorem 13.1 shows that SATPL is in NP. But SAT is a restriction of SATPL. The evidence checker for SATPL also works for SAT.

◇————————————————————————————————◇

Cook and Levin independently showed that SAT is NP-complete. The proof is too long for this course, so we will need to accept it as proved.

Theorem 15.2. (Cook/Levin Theorem) SAT is NP-complete.

That gives us evidence (but short of a proof, since it depends on the conjecture that $Px \neq$ NP) that there is no polynomial-time algorithm for SAT.

15.2 Proving NP-completeness

SAT is proved NP-complete using a difficult kind of reduction called a *generic reduction*. If all you know about X is that $X \in$ NP, you can ask someone to give you an evidence checker for X. The generic reduction from X to SAT converts that evidence checker into a propositioal formula.

But we don't need to do a generic reduction for every proof of NP-completeness.

Theorem 15.3. Suppose that language $B \in$ NP, A is NP-complete and $A \le_p B$. Then B is NP-complete.

Proof. Since $B \in$ NP, it suffices to show that $X \le_p B$ for every language $X \in$ NP. Since A is NP-complete, we know that $X \le_p A$ for every language $X \in$ NP. Since $A \le_p B$ and relation $\le_p$ is transitive (Theorem 14.3), $X \le_p B$ for every language $X \in$ NP.

◇————————————————————————◇

15.3 3-SAT

We can restrict the satisfiability problem further.

Definition 15.5. A propositional formula is in *3-clausal form* if it is in clausal form and has exactly 3 literals per clause. For example, $(x \vee y \vee z) \wedge (\overline{y} \vee z \vee \overline{w})$ is in 3-clausal form. A propositional formula in 3-clausal form is called a *3-clausal propositional formula*.

Definition 15.6. *3-SAT* is the following decision problem.

> **Input.** A 3-clausal propositional formula ϕ.
> **Question.** Is ϕ satisfiable?

Theorem 15.4. 3-SAT is NP-complete.

Proof. Clearly, 3-SAT is in NP. (Use the same evidence checker as for SATPL.) So, by Theorem 15.3, it suffices to reduce SAT to 3-SAT.

We need a polynomial-time algorithm that takes a clausal formula ϕ and builds a 3-clausal formula ϕ' such that ϕ is satisfiable if and only if ϕ' is satisfiable. Our algorithm will convert each clause of ϕ separately.

Clauses that already have 3 literals are left alone. Clauses with fewer than 3 literals are easy to deal with by duplicating one or more of the literals. For example, clause $(A \vee B)$ is equivalent to $(A \vee A \vee B)$.

That only leaves *long clauses*, which have more than 3 literals. As long as there is at least one long clause, we find one with n literals and replace it by a clause that has $n-1$ literals, plus a clause with 3 literals. By repeating that, we can get rid of all of the long clauses. It is just a matter of ensuring that each step preserves satisfiability.

Suppose that ϕ contains clause

$$C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n)$$

where $n > 3$. Create a new variable u and replace C by pair of clauses

$$C' = (\ell_1 \vee \cdots \vee \ell_{n-2} \vee u) \wedge (\overline{u} \vee \ell_{n-1} \vee \ell_n)$$

yielding new formula ϕ_1.

Claim 1. If ϕ_1 is satisfiable then ϕ is satisfiable. In fact, every truth-value assignment that satisfies ϕ_1 also satisfies ϕ.

Proof of Claim 1. Suppose ϕ_1 is satisfiable. Choose a truth-value assignment a that makes ϕ_1 true. That assignment must make all of the clauses other than C in ϕ true, since those clauses also occur in ϕ_1. We just need to argue that assignment a also makes clause C true.

Suppose $a(u) = \mathrm{F}$. Then, because a makes clause $(\ell_1 \vee \cdots \vee \ell_{n-2} \vee u)$ true, a must make at least one of $\ell_1, \ldots, \ell_{n-2}$ true. But that means a makes clause C true.

Suppose that $a(u) = \mathrm{T}$. Then, because a makes clause $(\overline{u} \vee \ell_{n-1} \vee \ell_n)$ true, a makes at least one of ℓ_{n-1} and ℓ_n true. Again, a makes clause C true.

Claim 2. If ϕ is satisfiable then ϕ_1 is satisfiable.

Proof of Claim 2. Suppose that ϕ is satisfiable, and choose a truth-value assignment a that makes ϕ true. Clause C must have at least one true literal.

If a makes ℓ_i true where $i \le n-3$, then extend a by adding $u = \mathrm{F}$. The new truth-value assignment makes clause $(\ell_1 \vee \cdots \vee \ell_{n-2} \vee u)$ true because ℓ_i is true, and it makes clause $(\overline{u} \vee \ell_{n-1} \vee \ell_n)$ true because $u = \mathrm{F}$.

If a makes ℓ_i true where $i > n - 3$, then extend a by adding $u = \mathrm{T}$. You can check that both clauses of C' must be true.

◇————————————————————————————◇

15.4 The Vertex Cover Problem

Recall from Section 13.4.2 that a vertex cover of a simple graph is a set C of vertices so every edge is incident on at least one vertex in C. VCP is the following decision problem.

Input. A simple graph G and a positive integer k.
Question. Does there exist a vertex cover C of G where $|C| \le k$?

It is worth asking whether there is an obvious polynomial-time algorithm for VCP. One idea is to use a *greedy algorithm*, which tries to optimize globally by optimizing locally. Since we want to select as few vertices as possible to cover all of the edges, it makes sense to start by selecting a vertex with highest degree, since it covers as many edges as possible with the first pick. After that, remove the selected vertex and all of the edges that it covers, and repeat, again selecting a vertex with the highest degree.

That algorithm seems appealing, but does it work? Look at graph G_1 in Figure 15-1. It has a vertex of degree 4, and all other vertices have degree 2 or 3. But the degree 4 is not part of any smallest vertex cover of G_1! If you are trying to determine whether G_1 has a vertex cover of size at most 4, you will be led astray by selecting the degree 4 vertex. Something is wrong with the greedy Vertex Cover algorithm.

It is tempting to try to patch the greedy algorithm. But is that worthwhile? The following theorem shows that it is a waste of time.

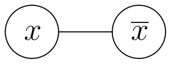
Theorem 15.5. VCP is NP-complete.

If the conjecture P $\neq$ NP is true then there does not exist a polynomial-time algorithm for VCP. Even if the conjecture is wrong, finding a polynomial-time algorithm for VCP is as difficult as proving that P = NP, since the existence of such an algorithm implies P = NP.

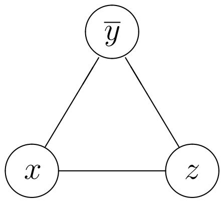
Proof of Theorem 15.5 Section 13.4.2 shows that VCP is in NP. We only need to reduce a known NP-complete problem to VCP. Let's show that 3-SAT $\le_p$ VCP.

We need a polynomial-time algorithm that takes a propositional formula ϕ in 3-clausal form and builds a pair consisting of a simple graph G and a positive integer k, where ϕ is satisfiable if and only if G has a vertex cover of size at most k.

The first step is construction of G. There are three parts. Part 1 consists of a pair of vertices for each variable that occurs in ϕ, which we call a *vertex gadget*. If x is a variable, add the following, where one vertex is labeled x and the other is labeled $\overline{x}$.



Part 2 consists of three vertices for each clause of ϕ, all connected to one another and labeled by the three literals in the clause, which we call a *clause gadget*. For clause $(x \vee \overline{y} \vee z)$, we add

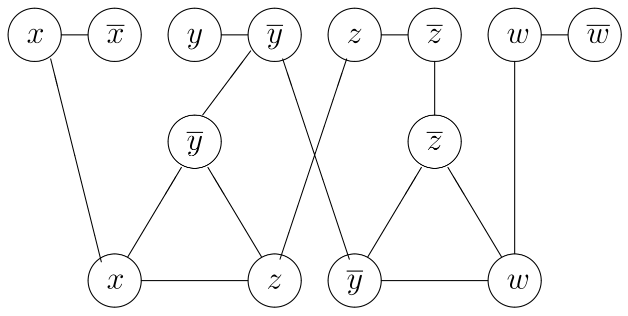


Part 3 does not add any vertices, but adds edges between part 1 vertices and part 2 vertices. Specifically, each part 2 vertex is connected to the part 1 vertex that has the same label.

Here is an example. Suppose

$$\phi = (x \vee \overline{y} \vee z) \wedge (\overline{y} \vee \overline{z} \vee w).$$

Then graph G looks like this:



That finishes the description of G. Suppose that ϕ has v variables and c clauses. Any vertex cover of G will need to have size at least $v + 2c$, one to cover each vertex gadget and two to cover each clause gadget. Let's choose $k = v + 2c$, not leaving any room for extra vertices in the vertex cover.

Claim 1. If ϕ is satisfiable then G has a vertex cover of size k.

Proof of Claim 1. Suppose ϕ is satisfiable, and let a be a truth-value assignment that makes ϕ true. Here is how to select a vertex cover of G of size k.

(a) For each variable x, if $a(x) = \text{T}$ then select the vertex gadget-vertex labeled x. Otherwise, select the vertex-gadget vertex labeled $\overline{x}$. That puts one vertex for each vertex gadget in the vertex cover, which covers the edges within vertex gadgets.

(b) For each clause $C = (\ell_1 \vee \ell_2 \vee \ell_3)$, find a literal ℓ_i that truth-value assignment a makes true. Select the clause-gadget vertices that correspond to the other two literals, leaving the literal labeled ℓ_i unselected. That covers all edges within clause gadgets.

There is no room to select any more vertices, but the part 3 edges between clause gadgets and vertex gadgets need to be covered. The unselected vertex u in a clause gadget corresponds to a true literal ℓ_i (under truth-value assignment a). A part 3 edge connects u to a vertex-gadget vertex v labeled ℓ_i, and, since ℓ_i is true, vertex v was selected, and edge $\{u, v\}$ is covered by v. No more vertices need to be added.

Claim 2. If G has a vertex cover of size k then ϕ is satisfiable.

Proof of Claim 2. Suppose G has a vertex cover S of size k. We know that S must select exactly one vertex from each vertex gadget and exactly two vertices from each clause gadget. Define truth-value assignment a so that $a(x) = \text{T}$ if the vertex-gadget vertex labeled x is in S, and choose $a(x) = \text{F}$ if the vertex-gadget vertex labeled $\overline{x}$ is in S.

Consider a clause gadget C. It has one vertex u that is not in vertex cover S. Suppose u is labeled by ℓ. There is an edge between u and a vertex-gadget vertex v that is also labeled ℓ. Since S is required to cover all edges, and S does not contain u, S must contain v. But truth-value assignment a is defined so that literal ℓ is true; if v is labeled x then $a(x) = \text{T}$, and if v is labeled $\overline{x}$ then $a(x) = \text{F}$, making $\overline{x}$ true. Therefore, the clause of ϕ that corresponds to clause gadget C has a true literal, namely ℓ.

The two claims show that the algorithm described above is a mapping reduction from 3-SAT to VCP. It should be obvious that the algorithm runs in polynomial time.

◇————————————————————◇

15.5 The Independent Set Problem

Definition 15.7. Suppose $G = (V, E)$ is a simple graph. An *independent set* of G is a set $S \subseteq V$ such that no two members of S are connected by an edge. That is, if u and v are different members of S, then$\{u, v\} \notin E$.

Definition 15.8. The *Independent Set Problem* (ISP) is the following decision problem.

> **Input.** A simple graph $G = (V, E)$ and a positive integer k.
> **Question.** Does G have an independent set of size at least k?

Look at graph G_1 in Figure 15-1. Some vertices are solid black and some are circles. Notice that the solid vertices are a vertex cover of G and the empty circles are an indenpendent set of G. Is that a coincidence? Think about it.

Theorem 15.6. Suppose $G = (V, E)$ is a simple graph and $S \subseteq V$. S is a vertex cover of G if and only if $\overline{S}$ is an independent set of G.

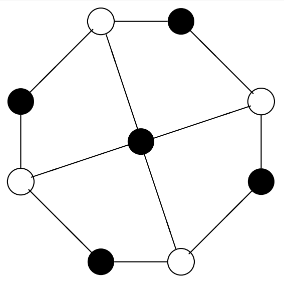


Figure 15-1 Graph G_1. The solid vertices are an independent set of G_1 and the empty circles are a vertex cover of G_1.

Proof. Suppose that $G = (V, E)$. Saying that S is a vertex cover of G is equivalent to the following logical statement.

$$\forall u \forall v(\{u, v\} \in E \rightarrow (u \in S \vee v \in S)).$$

Using the law of the contrapositive, that is equivalent to

$$\forall u \forall v(\neg(u \in S \vee v \in S) \rightarrow \{u, v\} \notin E).$$

Using DeMorgan's law and the definition of $\overline{S}$, that is equivalent to

$$\forall u \forall v((u \in \overline{S} \wedge v \in \overline{S}) \rightarrow \{u, v\} \notin E).$$

That is exactly what it means for $\overline{S}$ to be an independent set of G.

◇————————————————————————◇

Theorem 15.7. VCP $\le_p$ ISP.

Proof. Suppose that G has n vertices. For any set of vertices S of G, $|\overline{S}| = n - |S|$. That means $f(G, k) = (G, n-k)$ is a polynomial-time reduction from VCP to ISP, since

$$\begin{array}{rcl} (G,k) \in \text{VCP} & \leftrightarrow & G \text{ has a vertex cover of size at most } k \\ & \leftrightarrow & G \text{ has an independent set of size at least } n-k \\ & \leftrightarrow & (G, n-k) \in \text{ISP} \end{array}$$

◇————————————————————————◇

Corollary 15.8. ISP is NP-complete.

Proof. It is clear that ISP $\in$ NP. Theorem 15.7 shows that known NP-complete problem VCP polynomial-time reduces to ISP.

◇————————————————————◇

15.6 The Clique Problem

Another NP-complete problem about graphs is the Clique Problem.

Definition 15.9. Suppose that $G = (V, E)$ is a simple graph. A set $S \subseteq V$ is a *clique* if every pair of vertices in S are adjacent. That is, S is a clique if for all pairs of different vertices u and v in S, $\{u, v\} \in E$.

Definition 15.10. The *Clique Problem* (CP) is the following decision problem.

> **Input.** A simple graph G and a positive integer k.
> **Question.** Does G have a clique of size at least k?

Definition 15.11. Suppose $G = (V, E)$ is a simple graph. Then $\overline{G} = (V, \overline{E})$ is the complement of G, formed by complementing the set of edges. That is $\overline{G}$ has an edge between different vertices u and v if and only if G does not have an edge between u and v.

The following Theorem 15.9 is immediate from the definitions of independent sets and cliques.

Theorem 15.9. Suppose $G = (V, E)$ is a simple graph. S is an independent set of G if and only if S is a clique of $\overline{G}$.

Theorem 15.10. ISP $\le_p$ CP

Proof. By Theorem 15.9, function $f(G, k) = (\overline{G}, k)$ is a polynomial-time reduction from ISP to CP.

◇————————————————————◇

15.7 The Subset Sum Problem

The Subset Sum Problem is a generalization of the Partition Problem that we looked in Section NPSec.

Definition 15.12. The *Subset Sum Problem* (SSP) is the following decision problem.

> **Input.** A list $x_1, \ldots, x_n$ of positive integers and a positive integer K.
> **Question.** Does there exist an index set $I \subseteq \{1, \ldots, n\}$ so that
>
> $$\sum_{i \in I} x_i = K?$$

Theorem 15.11. SSP $\in$ NP.

Proof. The question in the definition of SSP is a question of existence. That suggests using I, the thing whose existence is questioned, as the evidence. Here is a polynomial-time evidence checker for SSP.

Evidence checker for SSP	
Input.	List $x_1, \ldots, x_n$ of positive integers and positive integer K
Evidence.	Index set $I \subseteq \{1, \ldots, n\}$
Requirement.	Is $\sum_{i \in I} x_i = K$?.

◇————————————————————◇

Theorem 15.12. 3-SAT $\le_p$ SSP.

Proof. Like the proof of Theorem 15.5, showing that 3-SAT $\le_p$ VCP, this proof requires some thought and some gadgetry. A polynomial-time reduction from 3-SAT to SSP is a polynomial-time computable function $f(\phi) = (L, K)$ where ϕ is a propositional formula in 3-clausal form, $L = x_1, \ldots, x_n$ is a list of positive integers and K is a positive integer so that ϕ is satisifiable if and only if $(L, K) \in$ SSP.

Writing a program for the reduction is not very informative. It is much easier to understand the reduction from an example. Suppose that ϕ is

$$(x \vee y \vee \overline{z}) \wedge (\overline{y} \vee z \vee \overline{w}) \wedge (w \vee \overline{x} \vee z)$$

	w	z	y	x	c_1	c_2	c_3
N_x:				1	1	0	0
$N_{\overline{x}}$:				1	0	0	1
N_y:			1	0	1	0	0
$N_{\overline{y}}$:			1	0	0	1	0
N_z:		1	0	0	0	1	1
$N_{\overline{z}}$:		1	0	0	1	0	0
N_w:	1	0	0	0	0	0	1
$N_{\overline{w}}$:	1	0	0	0	0	1	0
$P_{1,1}$:					1	0	0
$P_{1,2}$:					1	0	0
$P_{2,1}$:					0	1	0
$P_{2,2}$:					0	1	0
$P_{3,1}$:					0	0	1
$P_{3,2}$:					0	0	1
K:	1	1	1	1	3	3	3

Figure 15-2. List L consists of N_x and $N_{\overline{x}}$ for each variable x plus $P_{i,1}$ and $P_{i,2}$ for each clause c_i. Numbers are written in base 10. Notice that the sum can never involve a carry since there are no more than five 1s in any column.

with clauses c_1, c_2 and c_3. The result (L,K) of $f(\phi)$ is shown in Figure 15-2. List L is broken into two parts.

Part 1 of list L has two numbers N_x and $N_{\overline{x}}$ for each variable x. Think of those numbers written in base 10, with each number having two sections, the *variable section* and the *clause section*. The variable section has a column for each variable and the clause section has a column for each clause.

1. In the variable section, N_x and $N_{\overline{x}}$ each have a 1 in the column for x, with all other digits being 0.

2. In the clause section, N_x has a 1 in column c_i if x occurs in clause c_i, and it has a 0 in column c_i otherwise. Similarly, $N_{\overline{x}}$ has a 1 in column c_i if $\overline{x}$ occurs in clause c_i, and a 0 in column c_i otherwise.

Part 2 of list L has two numbers $P_{i,1}$ and $P_{i,2}$ for each clause c_i, which both contain only a 1 in the column that corresponds to c_i, as shown in Figure 15-2. They are used as *padding*

Claim 1. If ϕ is satisfiable then there is a way to select numbers from list L whose sum is K.

Proof of Claim 1. Suppose that a is a truth-value assignment that makes ϕ true. It tells which numbers to select to make a sum of K. First, select a true literal from each clause. If literal x is selected, put N_x into the list of selected numbers. If literal $\overline{x}$ is selected, put $N_{\overline{x}}$ into the list of selected numbers. If neither x nor $\overline{x}$ is selected, it does not matter; put N_x into the list of selected numbers.

Notice that the sum of the selected numbers has exactly one 1 in each column the variable section, so the part of K consisting of 1s is correct. We need to make sure the section of K consisting of 3s is correct. Because each clause contains a true literal, there must be at least one 1 in each clause column. But the total number of 1s in a single clause column in part 1 can be at most 3 since each clause contains 3 literals. If a clause column has one 1, then select both of the padding (part 2) numbers for that column to make a total of exactly 3. If there are two 1's, select one of the padding numbers. If there are three 1's, do not select any of the padding numbers for that clause.

The sum of the selected numbers is exactly K.

Claim 2. If it is possible to select numbers from list L whose sum is K then ϕ is satisfiable.

Proof of Claim 2. Because each variable column must sum to 1, exactly one of N_x and $N_{\overline{x}}$ must have been chosen for each variable x. Define truth-value assignment a so that $a(x) = \text{T}$ if N_x is selected and $a(x) = \text{F}$ if $N_{\overline{x}}$ is selected.

The selected numbers must sum to 3 in each clause column. At most two 1s in column i can come from padding numbers. The third must come from N_x, where x occurs in clause c_i, or from $N_{\overline{x}}$ where $\overline{x}$ occurs in clause c_i. That means c_i contains a true literal under truth-value assignment a.

The two claims show that the algorithm described above is a mapping reduction from 3-SAT to SSP. It should be obvious that the algorithm runs in polynomial time.

◇—————————————————————————◇

15.8 Graph Coloring Problems

Let's look at some known NP-complete problems without proving them NP-complete.

Definition 15.13. Suppose that G is a simple graph and k is a positive integer. Say that G is *k-colorable* if it is possible to color each vertex of G with one of k colors so that no two adjacent vertices have the same color.

Definition 15.14. The *Graph Coloring Problem* is the following decision problem.

Input. A simple graph G and a positive integer k.
Question. Is G k-colorable?

The Graph Coloring Problem is clearly in NP. The question asks whether *there exists* a way to color the vertices of G so that now two adjacent vertices have the same color. The obvious evidence to request is the coloring.

Evidence checker for Graph Coloring	
Input	Simple graph G and positive integer k.
Evidence	Assignment A of one of k colors to each vertex of G.
Requirement	Every edge of G connects two vertices that are assigned different colors in color assignment A.

If you try to color some graphs by hand, you can get an idea of how difficult Graph Coloring can be. The Graph Coloring Problem is known to be NP-complete. In fact, it is NP-complete even if k is fixed at 3.

Definition 15.15. The *3-Coloring Problem* is the following decision problem.

Input. A simple graph G.
Question. Is G 3-colorable?

Graph Coloring is so difficult, it can even be restricted further and yet remain NP-complete. Recall that a graph is *planar* if it can be drawn in the plane (on a piece of paper, if you like) so that no two edges cross one another.

Definition 15.16. The *Planar 3-Coloring Problem* is the following decision problem.

> **Input.** A planar simple graph G.
> **Question.** Is G 3-colorable?

The Planar 3-Coloring Problem is NP-complete. But that does not mean that all graph coloring problems are NP-complete. For example, 2-coloring is easy. Also, if G is known to be a planar graph, then 4-coloring is trivial: the answer is always yes, by the following.

Theorem 15.13. (The 4-Color Theorem) Every planar graph is 4-colorable.

15.9 Hamilton Cycles and Hamilton Paths

Definition 15.17. Suppose that G is a simple graph. A *simple cycle* in G is a cycle that does not contain any vertex more than once. A *Hamilton Cycle* is a simple cycle that contains every vertex.

Not every graph has a Hamilton cycle. You should be able to find a graph that has a Hamilton cycle and another that does not.

Definition NPCExampleSec.18. The *Hamilton Cycle Problem* is the following decision problem.

> **Input.** A simple graph G.
> **Question.** Does G have a Hamilton cycle?

Imagine that G has been drawn on paper (possibly with edges crossing). The Hamilton Cycle Problem asks whether it is possible to draw a cycle, following the edges, that hits every vertex exactly once, without lifting your pencil off the paper.

It is easy to show that the Hamilton Cycle Problem is in NP. The obvious evidence is a Hamilton cycle.

Evidence checker for Hamilton Cycle	
Input	Simple graph G with n vertices.
Evidence	Sequence $v_1, \ldots, v_n$ of vertices of G.
Requirement	$v_1, \ldots, v_n$ contains every vertex exactly once, $v_1 = v_n$ and for $i = 1, \ldots, n-1$, $\{v_i, v_{i+1}\}$ is an edge of G.

The Hamilton Cycle Problem is known to be NP-complete. A related problem, also NP-complete, is the Hamilton Path Problem.

Definition 15.19. Suppose that G is a simple graph. A *simple path* in G is a path that does not contain any vertex more than once. A *Hamilton Path* is a simple path that contains every vertex (exactly once).

Definition NPCExampleSec.20. The *Hamilton Path Problem* is the following decision problem.

> **Input.** A simple graph G.
> **Question.** Does G have a Hamilton path?

15.9.1 Euler Cycles

A problem that is at least superficially related to the Hamilton Cycle Problem is the Euler Cycle Problem. (Leonard Euler's last name is pronounced "Oiler".)

Definition NPCExampleSec.21. Suppose that G is a simple graph. An *Euler Cycle* in G is a cycle that contains each *edge* exactly once.

Not every graph has an Euler cycle. You should be able to find a graph that has an Euler cycle and another that does not.

Definition NPCExampleSec.22. The *Euler Cycle Problem* is the following decision problem.

> **Input.** A simple graph G.
> **Question.** Does G have an Euler cycle?

How difficult is it to determine whether a graph contains an Euler cycle? It is easy to see that the Euler Cycle Problem is in NP.

Evidence checker for Euler Cycle	
Input	Simple graph G with n vertices.
Evidence	Sequence $v_1, \ldots, v_n$ of vertices of G.
Requirement	$v_1 = v_n$, for $i = 1, \ldots, n-1$, $\{v_i, v_{i+1}\}$ is an edge of G (so $v_1, \ldots, v_n$ is a cycle) and cycle $v_1, \ldots, v_n$ uses each edge exactly once.

So we have an *upper bound* on the difficulty of solving the Euler Cycle Problem: it is in NP, so is, to within a polynomial, no worse than SAT. But that is not a *lower bound*. It might be that the Euler Cycle Problem is easy to solve.

And in fact, it is. Graph G has an Euler cycle if and only if every vertex has even degree. That is easy to check. Not only is the Euler Cycle Problem in P, but it is solvable in time $O(n)$.

There is a lesson in that. You cannot inspect a problem and conclude, based on its similarity to another problem, that it is an easy or a difficult problem. To show that a problem is in P, find a polynomial-time algorithm for it, and *make sure that the algorithm works.* To show that a problem is NP-complete, show that it is in NP and that a known NP-complete problem reduces to it in polynomial time. There are no shortcuts.

Examples of NP-complete problems

16 Beyond NP

The theory of NP-completess is a bedrock of computer science because there are so many NP-complete problems, and they crop of everywhere. There are NP-complete problems from mathematics, the theory of databases, the theory of compilers and even from politics.

But even though NP-completeness is central, there is more to the world than that. This section looks at two other classes of problems.

16.1 Co-NP and the validity problem

We started looking at difficult problems in Section 12.3 with the Validity Problem for Propositional Logic (VALIDPL). But we have not said anything more about it yet. We have not shown that it is NP-complete, nor have we shown that it is in P.

There is a good reason for that. If Conjecture 15.1 is true and P $\neq$ NP, then the validity problem is neither in P nor NP-complete. That is a consequence of the asymmetry of NP: if $A \in$ NP, then there are short, easily checkable proofs that things are in A, but there is no requirement that there are short, easily checkable proofs that things are not in A.

But VALIDPL has short, easily checkable proofs of nonmembership. To show that ϕ is not valid, show that $\neg\phi$ is satisfiable by finding a truth-value assignment that makes $\neg\phi$ true. The following is obvious.

Theorem 16.1. Suppose ϕ is a propositional formula. ϕ is valid if and only if $\neg\phi$ is not satisfiable.

Define $\overline{\text{SATPL}}$ to be the set of propositional formulas that are not satisfiable. Then $f(\phi) = \neg\phi$ is a polynomial-time reduction from VALIDPL to $\overline{SATPL}$. The same function is a polynomial-time reduction from $\overline{SATPL}$ to VALIDPL. So VALIDPL is equivalent in difficulty to $\overline{SATPL}$.

What can we say about languages that are complements of NP-complete languages?

Definition 16.1. Co-NP $= \{X \mid \overline{X} \in \text{NP}\}$ is the class of all languages whose complements are in NP.

Pay close attention to the definition of Co-NP. Co-NP is not the complement of NP.

Definition 16.2. Language A is *Co-NP-complete* if $A \in$ Co-NP and $X \leq_p A$ for every language $X \in$ Co-NP.

Theorem 16.2. A is NP-complete if and only if $\overline{A}$ is Co-NP-complete.

Proof. By the definition of Co-NP, $A \in$ NP $\rightarrow \overline{A} \in$ Co-NP and $\overline{A} \in$ NP $\rightarrow$ $\overline{\overline{A}} \in$ Co-NP. But $\overline{\overline{A}} = A$.

Suppose that A is NP-complete. Then $X \leq_p A$ for every X in NP. Suppose that f is a polynomial-time reduction from X to A. By the definition of a polynomial-time reduction,

$$x \in X \leftrightarrow f(x) \in A.$$

So

$$x \notin X \leftrightarrow f(x) \notin A.$$

or, equivalently,

$$x \in \overline{X} \leftrightarrow f(x) \in \overline{A}.$$

So f is a polynomial-time reduction from $\overline{X}$ to $\overline{A}$. So $\overline{A}$ is Co-NP-complete.

The other direction, showing that if $X \in$ Co-NP then then $\overline{X}$ is NP-complete, is true by symmetry.

◊————————————————————————◊

Since VALIDPL is equivalent to $\overline{SATPL}$, VALIDPL is Co-NP-complete.

16.2 NP ∩ Co-NP and factoring

We know that P $\subseteq$ NP. By symmetry, P $\subset$ Co-NP. So P $\subset$ NP $\cap$ Co-NP. An obvious question is: Is P $=$ NP $\cap$ Co-NP. That is conjectured to be false.

Conjecture 16.1 P $\neq$ (NP $\cap$ Co-NP).

Proving Conjecture 16.1 is difficult because, if P $=$ NP then P $=$ Co-NP and the conjecture is false.

But what would lead people to make Conjecture 16.1? There must be some decision problem that is conjectured to be in NP $\cap$ Co-NP but not in P. And there is: factoring integers. Quick, what are the prime factors of 109,938,432,277?

The problem of factoring a given integer cannot be in NP $\cap$ Co-NP because it is not a decision problem; the result is a list of factors. But there is a decision problem that is equivalent to that in computational difficulty.

Definition 16.3. FACTOR is the following decision problem.

> **Input.** Two positive integers x and k.
> **Question.** Does there exist a factor y of x where $1 < y < k$?

If you have a polynomial-time algorithm that finds the factors of an integer then it is easy to decide FACTOR. And if you have a polynomial-time algorithm for FACTOR then you can find the smallest factor of an integer using binary search. Having found the smallest factor, you divide x by that factor and continue finding factors, stopping when the number that you have is prime. (As mentioned in Section 12, there is a known algorithm that determines whether a given integer is prime.)

Conjecture 16.2 FACTOR $\in (NP \cap \text{Co-NP}) - \text{P}$.

16.3 Public key cryptograpy

Cryptographic systems are based on keys. To encipher a message you use the encipher key, and to decipher a message you use the associated decipher key. A person who has the decipher key is said to decipher a message. A person who attempts to do the same job without the benefit of the decipher key is said to *decrypt* the message.

Traditional cryptography is based on the idea that someone attempting to decrypt an enciphered message does not have enough *information* to do so. The standard traditional cryptosystem is a one-time pad, where a randomly chosen and agreed on text or number is used as a key, which must only be used once without giving away information to an adversary. With a one-time pad, the enciper key and decipher keys are the same.

Public key cryptography takes a different viewpoint. The encipher and decipher keys are different, and the strength of the system is based on the idea that someone trying to decrypt a message does not have enough *time*. And that is based on the problem of decrypting a message being a computationally difficult problem.

Definition 16.4. A public key cryptosystem is described by two functions $E(k, x)$ and $D(j, y)$ where k is a public encipher key and j is a private decipher key, such that

1. For every x in a limited range, $D(j, E(k, x)) = x$ and $E(k, D(j, x)) = x$. That is, deciphering an enciphered message gives the original message, and enciphering a deciphered message also gives the original message.

2. There is a polynomial-time algorithm to compute $E(k, x)$ and another to compute $D(j, y)$.

3. The *decryption function* $D(y)$ is defined to take y and yield a value x such that $E(k, x) = y$; $D(y)$ decrypts without the benefit of j. There is no polynomial-time algorithm that computes $D(y)$.

It is important to realize that the encipher key k is public, available to anyone. The strength of a public key cryptosystem is tied to the (at least apparent) computational difficult of computing $D(y)$.

Several public key cryptosystems are known, and it is not our concern here to describe one. Rather, let's ask whether a public-key cryptosystem exists. Assume that function pair $(E(k, x),\ D(j, y))$ is such a cryptosystem. For simplicity, assume that messages (x and y) are integers. Text can always be encoded using integers.

Consider the following decision problem DECRYPT determined by the decryption function $D(y)$.

$$\text{DECRYPT} = \{(y, i) \mid D(y) < i\}.$$

DECRYPT must be in NP. As evidence, use the decipher key j. First compute $x = D(j, y)$, then compute $z = E(k, x)$. Accept j as evidence that $(y, i) \in \text{DECRYPT}$ provided $z = y$ and $x < i$. Requirement $z = y$ tells you that j is the correct decipher key and requirement $x < i$ tells you that $D(y) < i$.

DECRYPT must also be in Co-NP. The complement of DECRYPT is equivalent to language $\{(y, i) \mid D(y) \geq i\}$, and almost the same evidence checker works for that.

So DECRYPT is in NP $\cap$ Co-NP. But if DECRYPT is in P then there is a polynomial-time algorithm to compute $D(y)$. Simply use binary search to search for the smallest i such that $D(y) < i$. Then $D(y) = i - 1$. That leads to the following conclusion.

Theorem 16.3. A public key cryptosystem can only exist if P $\neq$ NP $\cap$ Co-NP.

It is no accident that public key cryptosystems are based on factoring or on other problems that are in NP $\cap$ Co-NP.

16.4 Polynomial Space

Definition 16.5. *PSPACE* is the class of all decision problems that can be solved using $O(n^k)$ bits of memory for some fixed k, where n is the length of the input.

It is known that NP $\subseteq$ PSPACE and Co-NP $\subseteq$ PSPACE, and it is conjectured that NP $\neq$ PSPACE (and Co-NP $\neq$ PSPACE). Polynomial space allows a lot of room for computations. Typical exponential-time algorithms only use a polynomial amount of memory.

A PSPACE-complete problem is one of the hardest problems in PSPACE.

Definition 16.6. A decision problem A if *PSPACE-complete* if

(a) A is in PSPACE and

(b) $X \leq_p A$ for every problem $X \in$ PSPACE.

PSPACE is closely related to computations where quantifiers alternate between universal and existential. For that reason, several PSPACE-complete problems are related to two-person games. The following are some PSPACE-complete problems.

Definition 16.7. *Generalized Checkers* is the following decision problem.

> **Input.** A placement of red and black kings on an $n \times n$ checkerboard.
> **Question.** Assuming that it is red's move, does red have a winning strategy from the given configuration?

Theorem 16.4. Generalized Checkers is PSPACE-complete.

Geography is a game that children can play without any props. A child thinks of the name of a country (or other chosen category). If the first child select Sweden, then the next child must select a country name that begins with N, the last letter of Sweden. Suppose that child chooses Nepal. Now the next player must choose a country name that starts with L. Countries cannot be reused, and the first child who cannot think of a country name loses.

There is a version of Geography that is played on a directed graph. A vertex is selected as the start vertex s. The first player selects a vertex u where there is a directed edge from s to u. The second player selects a vertex v where there is a directed edge from u to v. Play alternates. No vertex that was previously selected can be selected again.

Definition 16.7. The *Generalized Geography* problem is the following decision problem.

> **Input.** A directed graph G with a selected start vertex.
> **Question.** Does the first player have a winning strategy on the game of Geography played on G?

Theorem 16.5. Generalize Geography is PSPACE-complete.

In practice, PSPACE-complete problems appear very difficult to solve. If the conjecture that NP $\neq$ PSPACE is true, then PSPACE-complete problems do not have polynomial-time evidence checkers.

Surprisingly, however, nobody knows whether P = PSPACE. Even the huge jump from polynomial time to polynomial space is not enough for us to demonstrate a separation. If you want to prove that P $\neq$ NP, you might warm up by proving that P $\neq$ PSPACE; that ought to be easier.

16.5 Exponential time

Definition 16.8. *EXPTIME* is the class of decision problems that are solvable in time $O(2^{n^k})$ for some fixed k, where n is the length of the input.

Notice that EXPTIME allows an amount of time that is two to a polynomial in n. It is known that PSPACE $\subseteq$ EXPTIME, and PSPACE is conjectured to be a proper subset of EXPTIME.

It is known that P $\neq EXPTIME$. So at least one of the subset relations P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME is surely a proper subset relation. Of course, they are all conjectured to be proper.

There is a notion of an EXPTIME-complete problem, defined in the usual way as a hardest problem in EXPTIME.

Definition 16.9. A decision problem A is *EXPTIME-complete* if

(a) A is in EXPTIME and

(b) $X \leq_p A$ for every problem $X \in$ EXPTIME.

TRhere are EXPTIME-complete problems. There is a typed programming language called ML. The ML Type Checking problem is as follows.

Definition 16.10. The *ML Type Checking Problem* is the following decision problem.

Input. An ML program p.
Question. Is p well-typed (free of type errors)?

The ML Type Checking Problem is known to be EXPTIME-complete. Fortunately, ML programmers tend not to write programs that are difficult to type check. But if you want to, you can write an ML program that will bring an ML compiler to its knees; in practice, the memory requirements overwhelm the compiler, and it gives up. Beyond NP

17 Practice Questions

Table of contents

Practice question sets

Due 8/12

1. Fully parenthesize propositional formula $P \wedge Q \vee R$. That is, add parentheses so that the structure is determined by the parentheses without the need for rules of precedence.

2. Fully parenthesize propositional formula $P \wedge Q \to R \wedge S$.

3. Fully parenthesize propositional formula $P \to Q \to R$.

4. How many rows are in the truth table of propositional formula $P \to (Q \to (\neg R \to S))$?

5. Construct a truth table of $P \to (Q \wedge P)$.

6. Using a truth table, show that $P \to (Q \to P)$ is a tautology.

7. Using a truth table, show that $(P \wedge Q) \to R \equiv (P \wedge \neg R) \to \neg Q$ is a tautology.

8. Suppose that the domain of discourse is the set of all integers. Say whether each of the following is true or false.

   (a) $\forall n \exists m (n^2 < m)$.
   (b) $\exists m \forall n (n^2 < m)$.
   (c) $\exists n \exists m (n^2 + m^2 = 5)$.
   (d) $\exists n \exists m (n^2 + m^2 = 6)$.
   (e) $\forall n \forall m \exists r (m + n = 2p)$.

9. Are first-order formulas $\exists x P(x) \wedge \exists x Q(x)$ and $\exists x (P(x) \wedge Q(x))$ logically equivalent? (That is, is

$$\exists x P(x) \wedge \exists x Q(x) \equiv \exists x (P(x) \wedge Q(x))$$

valid?) If so, explain why. If not, give definitions of $P(x)$ and $Q(x)$ where they are different.

Due 8/14

1. Suppose that the domain of discourse is the set of integers. Prove that, if $k+n$ is even and $n+m$ is even then $k+m$ is even.
2. Suppose that the domain of discourse is the set of real numbers. Let $R(x)$ be defined to mean "x is a rational number." Prove

$$\exists x \exists y(\neg R(x) \wedge \neg R(y) \wedge R(xy))$$

.

3. Using proof by contradiction, prove that, if n is an integer where n^3+5 is odd, then n is even.

Due 8/17

1. Write an enumeration of the members of set $\{x \mid x \in \mathcal{Z} \wedge x \geq 0 \wedge x > x^2 - 5\}$.
2. Give an enumeration of the members of each of the following sets.
   (a) $\{1, 3, 5, 6\} \cup \{2, 3, 5, 9\}$
   (b) $\{1, 3, 5, 6\} \cap \{2, 3, 5, 9\}$
   (c) $\{1, 3, 5, 6\} - \{2, 3, 5, 9\}$
3. True or false?
   (a) $\{2, 4, 6\} \subseteq \{2, 4, 6, 8\}$.
   (b) $\{2, 4, 6\} \in \{2, 4, 6, 8\}$.
   (c) $S - S = \{\}$ for every set S.
   (d) $2 \in \{2\}$.
   (e) $\{\} \in \{\}$.
   (f) $\{\} \subseteq \{\}$.
   (g) The empty set is a language.
   (h) The empty string is a language.
   (i) Every language is finite.
   (j) Every alphabet is finite.
   (k) A string can be infinitely long.
   (l) Some languages contain the empty string.
   (m) Every language contains the empty string.
   (n) A nonempty set of languages is not a language.

Due 8/19

1. Draw a transition diagram for a FSM with alphabet $\{a, b\}$ that decides language $\{\text{"}aab\text{"}\}$.
2. Draw a transition diagram for a FSM with alphabet $\{a, b\}$ that decides the set of all strings that begin with aa.
3. Draw a transition diagram for a FSM with alphabet $\{a, b\}$ that decides the set of all strings that end on aa. Number the states and say what $\text{Set}(q)$ is for each state q.
4. Draw a transition diagram for a FSM with alphabet $\{a, b\}$ that decides the set of all strings that contain exactly three bs. Number the states and say what $\text{Set}(q)$ is for each state q.
5. Draw a transition diagram for a FSM with alphabet $\{a, b, c\}$ that decides the set of all strings that contain $cacab$ as a contiguous substring.

Due 8/21

1. Let $L_1 = \{x \mid x \in \{a, b\}^* \wedge x \text{ has the same number of } a\text{'s as } b\text{'s}\}$. Prove that L_1 is not regular.

Due 8/24

1. Let $L_2 = \{www \mid w \in \{a, b, c\}^*\}$. Prove that L_2 is not regular.

Due 8/26

1. Is $\{\}$ computable? Justify your answer.

2. A positive integer n is *perfect* if n is the sum of its proper divisors. For example, 6 is perfect because $6 = 1 + 2 + 3$. Show that the set of perfect integers is computable.

3. Let $B = \{n \mid n$ is a positive integer that can be expressed as the sum of two prime numbers$\}$. For example, $8 \in B$ since $8 = 5 + 3$. Show that B is computable.

4. Prove that every finite language is computable.

5. Give an example of an infinite computable set.

Due 8/28

1. What properties does program I need to have for I to be an interpreter (for the programming language that has been chosen as the standard one for programs)?

2. Consider the following definition of a "semi-computable" language over alphabet Σ. For any program p, define $\text{Acc}(p) = \{x \mid \text{Run}(p, x) \cong 1\}$. Say that language A is semi-computable if there exists a program p where $\text{Acc}(p) = A$.

   Is the definition of a semi-computable language equivalent to the definition of a computable language? That is, is it true that A is computable if and only if A is semi-computable? Justify your answer.

3. Let A be the decision problem:

   Input. Two FSMs M_1 and M_2, both with alphabet Σ.
   Question. Is $L(M_1) \cup L(M_2) = \Sigma^*$?

   Show that A is computable.

4. Let B be the following decision problem:

   Input. A polynomial p of degree 3 with integer coefficients in a single variable, x.
   Question. Does there exist a value of x for which $p = 0$?

   Show that B is computable.

5. Let INFINITE be the following decision problem
   Input. A FSM M.
   Question. Does M accept infinitely many strings?

   Show that INFINITE is computable. It is not necessary to go into details about how to solve clearly solvable problems about graphs.

Due 8/31

1. Are all infinite languages uncomputable? Justify your answer.
2. Suppose that A and B are languages. What is the definition of a Turing reduction from A to B?
3. Define

   $$\begin{aligned} L_1 &= \{p \mid \text{Run}(p,p)\downarrow\} \\ L_2 &= \{(p,x) \mid \text{Run}(p,x)\downarrow\} \end{aligned}$$

   Give a Turing reduction from L_1 to L_2.
4. Define

   $$\begin{aligned} L_1 &= \{p \mid \text{Run}(p,1)\downarrow\} \\ L_2 &= \{p \mid \text{Run}(p,1)\uparrow\} \end{aligned}$$

   Give a Turing reduction from L_1 to L_2.
5. Suppose that A and B are languages over alphabet Σ where B is computable and $A \subseteq B$. Is it necessarily true that A is computable? Justify your answer.

   Think this out. Suppose that $B = \Sigma^*$. What are the subsets of B?

Due 9/2

1. What is the definition of a mapping reduction from language A to language B?
2. What is the definition of $A \leq_p B$?
3. Suppose that A and B are both computable languages over alphabet Σ. Show that $A \leq_t B$.
4. Suppose that A and B are both computable languages over alphabet Σ where $B \neq \{\}$ and $B \neq \Sigma^*$. Show that $A \leq_m B$.
5. Suppose that $A \leq_t B$ and B is a regular language. Does that imply that A is also a regular language? Justify your answer.
6. Suppose that A is a language where $A \leq_t$ HLT. Can you conclude that A is uncomputable?
7. Define

$$\begin{aligned} L_1 &= \{p \mid \text{Run}(p,p)\downarrow\} \\ L_2 &= \{(p,x) \mid \text{Run}(p,x)\downarrow\} \end{aligned}$$

   Give a mapping reduction from L_1 to L_2.

Due 9/4

1. Show that language $\{p \mid \text{Run}(p, \texttt{"}aa\texttt{"}) \cong 0 \text{ and } \text{Run}(p, \texttt{"}bb\texttt{"}) \cong 1\}$ is not computable.

2. Let $A = \{p \mid \text{Run}(p, \texttt{"}bbb\texttt{"})\downarrow\}$. Give a mapping reduction from A to HLT.

3. Define $B = \{p \mid L(p) \text{ is a regular language}\}$. Is B computable? Justify your answer.

4. For the purposes of this exercise, assume that the output of a program is an integer. Suppose $A = \{p \mid \text{Run}(p, 0) \cong 5\}$ and $B = \{p \mid \text{Run}(p, 0) = 10\}$. Give a mapping reduction from A to B. Be sure that you know what properties the reduction needs to have before you start to describe the reduction.

Due 9/9

1. What is the definition of class P?
2. Suppose that L is a set of positive integers and suppose that there is an algorithm that takes an integer n and tells you whether $n \in L$ in time $O(n^2)$. Can you conclude that L is in P based on that? Explain why or why not.
3. Suppose that L is a set of strings, and suppose that there is an algorithm that takes a string x and tells you whether $x \in L$ in time $O(2^n)$, where $n = |x|$. Can you conclude that L is not in P based on that? Explain why or why not.
4. A *triangle* in a simple graph consists of three mutually adjacent vertices. Show that the problem of determining whether a simple graph contains a triangle is in P.

Due 9/11

1. What is the definition of NP?

2. Is $\{\}$ in NP?

3. Suppose that Σ is an alphabet. Is Σ^* in NP?

4. A *bijection* is a function that is one-to-one and onto. Two simple graphs $G = (V, E)$ and $H = (W, F)$ are *isomorphic* if $|V| = |W|$ and there is a bijection $f : V \to W$ such that, for every pair of vertices a and b in V, $\{a, b\} \in E \leftrightarrow \{f(a), f(b)\} \in F$.

   The *Graph Isomorphism Problem* (GIP) is the following decision problem.

   > **Input.** Simple graphs G and H.
   > **Question.** Are G and H isomorphic?

   Show that GIP is in NP.

5. Let DOUBLE-SATPL be the following decision problem.

   > **Input.** A propositional formula ϕ.
   > **Question.** Do there exist two different truth-value assignments a and b where $(a \dashv \phi) = \mathrm{T}$ and $(b \dashv \phi) = \mathrm{T}$? That is, can ϕ be made true by two different choices of the values of its variables?

   Show that DOUBLE-SAT is in NP.

Due 9/14

1. What is the definition of a polynomial-time mapping reduction from language A to language B?

2. What is the definition of notation $A \leq_p B$?

3. Suppose that $A \in \mathrm{P}$ and $A \leq_p B$. Can you conclude that $B \in \mathrm{P}$?

4. Suppose that $B \in \mathrm{P}$ and $A \leq_p B$. Can you conclude that $A \in \mathrm{P}$?

5. Suppose that $\mathrm{P} = \mathrm{NP}$. Show that, for every $A \in \mathrm{NP}$, $A \leq_p \{1\}$.

6. SATPL is the following decision problem.

   Input. A propositional formula ϕ.
   Question. Does there exist a truth-value assignment a where $(a \dashv \phi) = \mathrm{T}$? That is, is it possible to choose values for the propositional variables in ϕ so that ϕ is true?

   Show that SATPL $\leq_p$ DOUBLE-SATPL by giving a polynomial-time mapping reduction from SATPL to DOUBLE-SATPL. (DOUBLE-SATPL is defined above.) (**Hint.** Add an extra variable.)

7. Give an example of a decision problem that is not in NP. Justify your answer.

Due 9/16

1. What is the definition of an NP-complete problem?
2. Does there exist a decision problem that is not in $\mathrm{P} \cup \mathrm{NP}$? Justify your answer.
3. Show that DOUBLE-SATPL $\leq_p$ SAT. You are not required to give a polynomial-time mapping reduction from DOUBLE-SATPL to SATPL. But give an air tight argument that such a mapping reduction must exist.
4. Let A be the set of all natural numbers that are prime. Does there exist a polynomial-time mapping reduction from A to SAT?
5. Suppose B is in NP and A is NP-complete and $A \leq_p B$. Can you conclude that B is NP-complete?
6. Suppose that A is NP-complete and $A \subseteq B$. Can you conclude that B is NP-complete? Justify your answer.

Due 9/18

1. If P = NP, is SAT NP-complete?
2. Is SAT known to be NP-complete, or is SAT only conjectured to be NP-complete?
3. Is it known that SAT $\notin$ P?
4. Show that DOUBLE-SATPL is NP-complete. You can appeal to answers to prior exercises without repeating them.
5. The Hitting Set Problem (HSP) is as follows.

   Input. Positive integers N and K; and a list of sets x_1, …, x_m, where $x_i \subseteq \{1, \ldots, N\}$ for $i = 1, \ldots, m$.
   Question. Does there exist a set $S \subseteq \{1, \ldots, N\}$ where $|S| \le K$ and $x_i \cap S \neq \{\}$ for $i = 1, \ldots, m$. That is, S must contain at least one member of each set x_i.

   (a) Give a polynomial-time evidence checker for HSP. Be sure that it is correct for every input and that your description is clear and easy to understand.

   (b) Give a polynomial-time reduction from the the Vertex Cover Problem (VCP) to HSP. Be sure that the reduction is correct for all possible inputs. Describe the reduction in a clear, readable way. Be sure that I can find your definition of the reduction. Just words describing what the reduction might do are not adequate.

   (c) Are the results of parts (a) and (b) of this problem sufficient for you to conclude that HSP is NP-complete? Explain why or why not.

   (d) Does there exist a polynomial-time reduction from HSP to VCP? Either argue that there probably is no such reduction or explain why there must exist such a reduction.

Due 9/21

1. The Partition Problem (PP) is described in Section 13. Give a polynomial-time reduction from PP to the Subset Sum Problem.

2. See above for the definition of isomorphic graphs.

   ¡p¿Suppose that G and H are simple graphs. Say that H is isomorphic to a subgraph of G provided it is possible to remove zero or more vertices and zero or more edges from G and get a graph that is isomorphic to H. (When you remove a vertex v, you must also remove all edges that are incident on v.)

   The Subgraph Isomorphism Problem (SIP) is the following decision problem.

   > **Input.** Simple graphs G and H.
   > **Question.** Is H isomorphic to a subgraph of G?

   Prove that SIP is NP-complete. (**Hint.** Reduce from the Clique Problem.)

Due 9/23

1. What is the definition of Co-NP?

2. Assume that P $\neq$ NP. Is the Validity Problem for Propositional Logic NP-complete? Explain.

3. Give a polynomial-time reduction from the Hamilton Cycle Problem to the Subgraph Isomorphism Problem (above).

4. Suppose that a particular university chooses a set C of classes to offer in a given term and has N time slots in which to schedule classes. Each student selects a set of classses that he or she wants to take.

   The Class Scheduling Problem (CSP) is the following decision problem.

   > **Input.** Positive integer N, set of classes $C = \{c_1, \ldots, c_m\}$, and list of sets $s_1, \ldots, s_k$ where $s_i \subseteq C$ is the set of classes that student i wants to take.
   > **Question.** Does there exist a way to schedule classes into time slots so that no student wants to take two classes that are assigned to the same time slot?

   (a) Prove that CSP is in NP by giving a polynomial-time evidence checker for CSP.

   (b) Give a polynomial-time reduction from the Graph Coloring Problem (GCP) to CSP. (**Hint.** Think about what corresponds to a vertex, what corresponds to an edge and what corresponds to a color.)

Practice questions