UNCOMPUTABILITY IN CPSC421/501

JOEL FRIEDMAN

Contents

Date: Sunday 24th November, 2024, at 22:51.
Research supported in part by an NSERC grant.

This version is A DRAFT: I'm in the process of adding material and editing article as of August 2024. I'll post to the course website when I revise this article.

Main reference: In this article, [Sip] refer to the course textbook, *Introduction to the Theory of Computation* by Michael Sipser, 3rd Edition.

Prerequisities: This article assumes you are familiar with the material in Chapter 0 of [Sip]. In addition, we assume that you are you have seen some analysis of algorithms, including big-Oh and little-oh notation (e.g., $n\log_2 n + 3n + 5 = n\log_2 n + O(n)$). We briefly review the definitions of strings and languages (see Chapter 0 of [Sip]). Appendix B gives a few more details and examples.

Acknowledgement: I have learned from many of my students, TA's, and colleagues; some are acknowledged in footnotes, specifically Benjamin Israel (in the versions of these notes since roughly the early 2000's) and Yuval Peres and Sophie MacDonald (since Fall 2021). Amir Tootooni, a TA for the course (in Fall 2020 and 2021), made many helpful suggestions and corrections for the 2021 version.

1. The Main Goals of This Article

The first two weeks of CPSC 421/501 will be spent covering parts of Sections 1–7 of this article; most of the rest of this article will be covered later in the course.

One main goal of this article is to introduce some material that is typical of the level of difficulty of CPSC 421/501. In particular, this article includes some material of Chapters 4 and 5 of [Sip]; this material is more difficult than Chapters 1–3 of [Sip], where the course usually begins.

Another goal is to introduce some basic terminology used throughout this course: alphabets, strings, languages, and "descriptions" of various objects (integers, programs, etc.) as strings.

This article will explain (thereby "spoiling") one of the main surprises in this course: "self-referencing" combined with "negation" leads to some wonderful "paradoxes" and/or "contradictions," which — sometimes — prove interesting theorems (e.g., the Halting Problem is undecidable).

The article is a currently a draft; some corrections and additions, especially to the exercise section, will likely be made as we cover this material.

2. Cantor's Theorem

The point of this section is to prove the deservedly famous *Cantor's theorem*. In Section 4 we apply Cantor's theorem to produce an *unrecognizable language*; the rest of these notes gives material on unrecognizable languages beyond these notes.

A secondary goal is to review some notions regarding set theory and functions.

2.1. Naive Set Theory.

In this section we work with "naive set theory," which means that we work informally with the usual notion of a *set*, its *elements*, a *subset of a set*, etc.

The reason we call this *naive* is that you can easily use the ideas of set theory to produce a contradition: indeed, Russell's (most famous) Paradox arises from considering the set, T, of all sets that don't contain themselves, i.e.,

$$T = \Bigl\{ S \mid S \text{ is a set such that } S \notin S \Bigr\}.$$

One easily shows that both $T \in T$ and $T \notin T$ are impossible (see Section 5). This caused mathematicians to be more careful about set theory and to give collections of axioms regarding set theory that avoid Russell's Paradox. For now we ignore such problems; just trust that everything we do in this section is justifiable when working with any of the standard choices of the axioms of set theory.

We warn the reader that working with infinite sets — which we need to do in this article — has some subtleties that don't arise if you work exclusively with finite sets.

2.2. The Power Set.

If S is any set, then $\mathrm{Power}(S)$ refers to the set of all subsets of S.

Example 2.1. Let $S = \{a, b, c\}$. Then

$$\mathrm{Power}(S) = \mathrm{Power}\bigl(\{a,b,c\}\bigr) = \Bigl\{ \emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\} \Bigr\},$$

where $\emptyset$ denotes the empty set.

Remark 2.2. If S is a finite set, and $|S|$ denotes the cardinality (size) of S, then $|\mathrm{Power}(S)| = 2^{|S|}$. For this reason one sometimes uses 2^S to denote the power set of S. This is also the origin of the term *power set*.

2.3. Functions.

If A, B are sets, we will speak of a function $f \colon A \to B$, also called a *map of sets* and a *morphism of sets*). A is called the *domain* of f, and B the *range* or *codomain* of f. The *image of* f, denoted $\mathrm{Image}(f)$, is the subset of B given by

$$\mathrm{Image}(f) = \{ f(a) \,|\, a \in A \} \subset B.$$

[To formally define what is meant by a function $f \colon A \to B$, one works with certain kinds of *relations*. Namely, a *relation on* A, B is a subset of $A \times B$ (the *cartesian product* (or simply *product*) of A and B). A relation, $R \subset A \times B$, is a *function* if for each $a \in A$ there exists exactly one $b \in B$ such that $(a,b) \in R$. To any such R we associate the "function" $f \colon A \to B$, where for each $a \in A$, $f(a) \in B$ is the unique element of B such that $(a, f(a)) \in R$. Conversely, to each function

$f\colon A \to B$ we associate the relation R on A, B, i.e., the subset $R \subset A \times B$, given by $\{(a, f(a)) | a \in A\}$.]

Definition 2.3. Let A, B be sets, and $f\colon A \to B$ a function. We say that f is:

(1) *injective* (also known as *into*, *an injection*, *one-to-one*, and *a monomorphism*) if $a, a' \in A$ with $a \neq a'$ implies $f(a) \neq f(a')$;
(2) *surjective* (a.k.a. *a surjection*, *onto*, and *an epimorphism*) if for all $b \in B$ there is an $a \in A$ with $f(a) = b$;
(3) a *bijection* (a.k.a. *a bijection*, *a one-to-one correspondence*[1], and *an isomorphism*) if it is injective and surjective.

Remark 2.4. Let $f\colon A \to B$ be a function of finite sets. Then if $|A| < |B|$, you should be able to convince yourself that f cannot be a surjection; hence if $A \subset B$ is a *proper subset* (i.e., $A \neq B$), then f cannot be a surjection.

Remark 2.5. If B is an infinite set, and $A \subset B$ is a proper subset, then there may exist a bijection (which is therefore a surjection) $f\colon A \to B$. For example, the *natural numbers* and *non-negative integers* are, respectively, the two sets

$$\mathbb{N} = \{1, 2, 3, \ldots\} \quad \text{and} \quad \mathbb{Z}_{\geq 0} = \{0, 1, 2, \ldots\}.$$

Hence $\mathbb{N}$ is a proper subset of $\mathbb{Z}_{\geq 0}$. Yet there is a bijection $f\colon \mathbb{N} \to \mathbb{Z}_{\geq 0}$ given by $f(x) = x - 1$.

Remark 2.6 (The Pigeon Hole Principle)**.** Similarly to Remark 2.4, if A, B are finite sets with $|A| > |B|$, you should be able to convince yourself that no function $f\colon A \to B$ can be injective; this is often called the "Pigeon Hole Principle" (when you have more pigeons than birdhouses) or, somewhat obsoletely, the "Drawers Principle" (more pairs of socks than drawers in a dresser).

Example 2.7 (Example of the Pigeon Hole Principle)**.** Say that an academic department has at least 53 profs, and that each prof has a last name written in English beginning with a letter A–Z or a–z. Then some two profs have last names beginning with the same letter.

2.4. **Cantor's Theorem.**

Theorem 2.8 (Cantor's Theorem)**.** *Let S be a set, and $f\colon S \to \mathrm{Power}(S)$ be any function. Then f is not a surjection, i.e., some $T \in \mathrm{Power}(S)$ is not in the image of f. Specifically the set*

$$T = \{s \mid s \notin f(s)\} \tag{1}$$

is not in the image of f, i.e., there is no $t \in S$ such that $f(t) = T$.

Proof. Assume, to the contrary, that some $t \in S$ has $f(t) = T$. Either $t \in T$ or $t \notin T$: let us derive a contradiction in either case.

If $t \in T$, then by the definition of T, $t \notin f(t)$. But, by assumption, $f(t) = T$ so $t \notin f(t) = T$, which contradicts the assumption that $t \in T$.

Similarly, if $t \notin T$, then t does not sastisfy $t \notin f(t)$. Hence $t \in f(t) = T$, which contradicts $t \notin T$. □

[1]We warn the reader that in mathematics, a *correspondence from A to B* (without writing "one-to-one") is often defined to be a morphism $g\colon C \to A \times B$ for some set C, contrary to what the textbook [Sip] writes.

2.5. **Examples of Cantor's Theorem.**

Example 2.9. Let $S = \{1,2,3\}$ and $f\colon S \to \mathrm{Power}(S)$ be given by

$$f(1) = \{1,2,3\},\ \ f(2) = \emptyset,\ \ f(3) = \{1,2\}.$$

Then we easily check that T in Theorem 2.8 is $T = \{2,3\}$, which is visibly not in the image of f. Of course, $|S| = 3$, and $|\mathrm{Power}(S)| = 2^3 = 8$, so it is clear that any $f\colon S \to \mathrm{Power}(S)$ is not surjective.

Example 2.10. A department has 3 profs, $P = \{a,b,c\}$. It is given that (1) Prof. a thinks that everyone in the department is clever, (2) Prof. b thinks that no one in the department is clever, and (3) Prof. c thinks that they alone are clever (i.e., that c is clever, and a,b are not clever). This gives a map $\mathrm{ThinksIsClever}\colon P \to \mathrm{Power}(P)$, namely

$$\begin{aligned}\mathrm{ThinksIsClever}(a) &= \{a,b,c\}\\ \mathrm{ThinksIsClever}(b) &= \emptyset\\ \mathrm{ThinksIsClever}(c) &= \{c\}\end{aligned}$$

Hence

$$\begin{aligned}a &\in \mathrm{ThinksIsClever}(a)\\ b &\notin \mathrm{ThinksIsClever}(b)\\ c &\in \mathrm{ThinksIsClever}(c)\end{aligned}$$

The set T of profs who do not consider themself to be clever is $T = \{b\}$; we easily directly check that T is not in

$$\mathrm{Image}(\mathrm{ThinksIsClever}) = \Bigl\{\{a,b,c\},\emptyset,\{c\}\Bigr\}.$$

Remark 2.11. Example 2.9 is really a "toy example" of Cantor's theorem; it has no particular application. By contrast, Example 2.10 is much closer in spirit to our application of Cantor's theorem in Section 4 to produce an *unrecognizable* language: this example involves a set P that "has meaning," and a conceptual map $\mathrm{ThinksIsClever}\colon P \to \mathrm{Power}(P)$ that "has meaning" but makes Cantor's theorem more difficult to "unpack." Of course, both examples above involve a three-element set and its power set.

Remark 2.12. Usually Cantor's theorem is stated more concisely (e.g., for any set, S, there is no surjection $S \to \mathrm{Power}(S)$).[2]. Theorem 2.8 above is really the usual statement of Cantor's theorem plus the usual first step in its proof. The reason we state Theorem 2.8 as such is that (1) allows us to explicitly describe a subset of S that is not in the image of f, which we will need in Section 4.

[2]It is equivalent to say that the *cardinality* of $\mathrm{Power}(S)$ is strictly larger than that of S, but you have to know what is meant by the *cardinality* of an infinite set.

2.6. **Cantor's Theorem, Diagonalization, and Yes/No Tables.** Let us descirbe Example 2.10 in the following table:

Is $y \in \text{ThinksIsClever}(x)$, i.e., does x think that y is clever?	$y = a$	$y = b$	$y = c$
$x = a$	`yes`	`yes`	`yes`
$x = b$	`no`	`no`	`no`
$x = c$	`no`	`no`	`yes`

The set of profs who do not think of themself as clever is obtained by taking the diagonal elements:

Is $y \in \text{ThinksIsClever}(x)$?	$y = a$	$y = b$	$y = c$
$x = a$	`yes`		
$x = b$		`no`	
$x = c$			`yes`

and putting s into T if the corresponding "diagonal element" is `no`. Hence $T = \{b\}$ in this case.

In terms of yes/no tables, we can extend the table to include T:

Is $y \in \text{ThinksIsClever}(x)$?	$y = a$	$y = b$	$y = c$
$x = a$	`yes`		
$x = b$		`no`	
$x = c$			`yes`
	$\downarrow$	$\downarrow$	$\downarrow$
Does $x = y$ lie in T?	`no`	`yes`	`no`

This table gives another way to understand why Cantor's theorem is true: the $x = a$ row of this table tells us that ThinksIsClever(a) contains a, but T does not contain a; hence $T \neq \text{ThinksIsClever}(a)$; similarly for the $x = c$ row; for the $x = b$ row, ThinksIsClever(b) contains b but T doesn't, so $T \neq \text{ThinksIsClever}(b)$.

Considering the `yes`/`no` tables above shows that Cantor's theorem is an example of *diagonalization* argument; the term *diagonalization* is really an umbrella term for a number of mathematical techniques in a number of contexts (e.g., analysis, including ODE's and PDE's) that appeal to "diagonal" of a square gid as above.

3. Generalized Cantor's Theorem and Partial Information

Often we want to apply the idea of Cantor's theorem to a function $f : S \to \text{Power}(B)$ where S, B are not the same set, but where B is "of the same size or larger than" S. [We use B to suggest "big," and S "small," although we may have $B = S$ or $B \simeq S$.[3]] Then it is still true that f cannot be a surjection, and we can demonstrate a set that is not in the image of f using an easy generalization of Cantor's theorem.

3.1. **Larger Sets: Some Subtleties.**

Note that if S, B are finite sets, then the following are equivalent:

(1) $|B| \geq |S|$;
(2) there exists a surjection $f : B \to S$; and
(3) there exists an injection $g : S \to B$.

[3]The notation $B \simeq S$ means that B and S are *isomorphic*, i.e., there exists a bijection $B \to S$.

Set theory has a bunch of subtleties; for example, if S, B are arbitrary (possibly infinite) sets, it does not seem obvious (to us) that (2) holds iff (3) holds, unless you assume the "Axiom of Choice."

[In slightly more detail: for arbitrary sets B, S, we believe that (3) implies (2) in the usual (ZF or Zermelo–Fraenkel) axioms of set theory[4], and that (2) implies (3) in ZFC, i.e., the usual ZF axioms plus assuming the Axiom of Choice [5]; we (due to our ignorance) don't know if (2) implies (3) without the Axiom of Choice.[6]

3.2. **Larger Sets: Definitions.**

Definition 3.1. If S, B are arbitrary sets, we write $|B| \geq |S|$ (and also $|S| \leq |B|$) if there is an injection from S to B.[7] We will write $|B| = |S|$ if there is a bijection $S \to B$, and otherwise write $|B| \neq |S|$; we will write $|B| > |S|$ if $|B| \geq |S|$ and $|B| \neq |S|$.

By Subsection 3.1, $|B| \geq |S|$ implies that there is a surjection $B \to S$.

In view of (1) and (3) of Subsection 3.1, our definition of $|B| \leq |S|$ coincides with the usual meaning of $|B| \leq |S|$ if S, B are finite sets. You should be able to convince yourself that if S is finite and B is infinite, then $|B| > |S|$; it should make sense that the "size" of an infinite set is larger than that of any finite set.

Example 3.2. There is an injection $f \colon \mathbb{Z}_{\geq 0} \to \mathbb{N}$ given by $f(x) = x + 1$; another injection $h \colon \mathbb{Z}_{\geq 0} \to \mathbb{N}$ is given by $h(x) = 3x + 2024$. Hence $|\mathbb{N}| \geq |\mathbb{Z}_{\geq 0}|$ (even though $\mathbb{N}$ is a proper subset of $\mathbb{Z}_{\geq 0}$). The function f is a bijection, and hence we write $|\mathbb{N}| = |\mathbb{Z}_{\geq 0}|$.

3.2.1. *More Examples.*

On class on September 9, 2024, we gave some examples that are covered later on in this article; much of this material is due to questions asked in class.

Example 3.3. Let $S \subset B$, i.e., S, B are sets and S is a subset of B (possibly all of B). Then there is an injection $S \to B$ mapping $s \in S$ to $s \in B$. Hence $|S| \leq |B|$. For example, if $S = \{1, 2\}$ and $B = \{1, 2, 3\}$, then the map $f \colon S \to B$ given by $f(1) = 1$ and $f(2) = 2$ is an injection.

Example 3.4. Let $\mathbb{Q}_+$ be the set of positive rational numbers, and $\mathbb{N} = \{1, 2, 3, \ldots\}$ be the natural numbers. Then we claim $|\mathbb{Q}_+| \leq |\mathbb{N}|$, and, moreover, $|\mathbb{Q}_+| = |\mathbb{N}|$. Indeed, every rational number appears on the list:

$$1/1, \quad 2/1, 1/2, \quad 3/1, 2/2, 1/3, \quad , \ldots \tag{2}$$

[4]Namely, if $f \colon B \to S$ is any injection, then for any $b' \in B$, we can define a surjection $g_{b'} \colon B \to S$ by setting $g_{b'}(b) = b'$ if b is not in the image of f, and for any b in the image of f, hence $b = f(s)$ for a unique s, we set $g_{b'}(b) = s$. [The fact that $b' \in B$ exists follows from the Axiom of Regularity in ZF.]

[5]Given a surjection $g \colon B \to S$, we can build an injection $f \colon S \to B$ by choosing for each $s \in S$ an element $b \in B$ such that $g(b) = s$ and defining $f(s) = b$; this "choosing" seems to require the Axiom of Choice.

[6]Set theory has a number of subtleties; for example, the Schröder–Bernstein Theorem shows that (under ZF, without assuming the Axiom of Choice), that if there are injections $S \to B$ and $B \to S$ then there is a bijection $S \to B$. However, there are sets A, B such that it is easy to describe injections $A \to B$ and $B \to A$ but more difficult to describe a bijection $A \to B$; see Exercise **??**. It is also known that ZFC, i.e., ZF plus the Axiom of Choice, implies that there is a good theory of cardinal numbers, and hence for any sets A, B, either $|A| \leq |B|$ or $|A| \geq |B|$.

[7]Typo corrected Sept 11; thanks to a CPCS 501 student.

(see Section 4.2 of [Sip]). By erasing repeated rational numbers from this list, we get a list

$$1/1, \quad 2/1, 1/2, \quad 3/1, \quad , 1/3, \quad 4/1, 3/2, 2/3, 1/4, \quad 5/1, \quad , 1/5, \quad \ldots \tag{3}$$

$(4/2, 3/3, 2/4$ are all repeats), which gives a bijection $\mathbb{Q}_+ \to \mathbb{N}$.[8]

Definition 3.5. We say that an infinite set, S, is *countably infinite* if there is a bijection $S \to \mathbb{N}$.

Hence $\mathbb{Q}_+$ is countably infinite.

In class we also gave the definitions in Subsection 4.1 and explained why for any alphabet, Σ, the set Σ^* is countably infinite.

For any set, S, it is easy to give an injection $S \to \mathrm{Power}(S)$: namely the map $s \mapsto \{s\}$. It follows that $|S| \le |\mathrm{Power}(S)|$; Cantor's theorem implies that there is no bijection $S \to \mathrm{Power}(S)$, and hence $|S| < |\mathrm{Power}(S)|$. Moreover, the statement "for any set S, $|S| < |\mathrm{Power}(S)|$" is the usual way that Cantor's theorem is stated.

3.3. Generalized Cantor's Theorem, Injective Form.

Theorem 3.6 (Injective Form of Generalized Cantor's Theorem)**.** *Let $h\colon S \to B$ be an injection. Then for any map $f\colon S \to \mathrm{Power}(B)$, the image of f is not all of* $\mathrm{Power}(B)$*; specifically*

(1) the set

$$T = \{h(s) \mid s \in S \text{ and } h(s) \notin f(s)\} \subset B \tag{4}$$

is not in the image of f;

(2) more generally, any set $T \subset B$ such that

$$\forall s \in S \qquad h(s) \in T \quad \iff \quad h(s) \notin f(s) \tag{5}$$

is not in the image of f (we say "more generally" because an example of set T satisfying (5) is given in (4)).

Proof. First we prove (2): if such a $T \subset B$ is in the image of f, then for some $t \in S$ we have $f(t) = T$. Then we have

$$h(t) \in T \quad \iff \quad h(t) \notin f(t) = T,$$

and hence $h(t) \in T \iff h(t) \notin T$, which is impossible. Using the fact that h is an injection, we easily see that (4) satisfies (5). □

In Theorem 3.6, note that (5) only specifies which elements of $\mathrm{Image}(h)$ are in or not in T; hence if the image of h is not all of B, there are at least two possible sets T satisfying (5).

[8]The number of rational numbers that are not repeats when you write n terms of (3) is well-known to be equal to $nc + o(n)$ where $c = 1/\zeta(2) = 6/\pi^2$. It will easily follow that there is no DFA (see Chapter 1 of [Sip]) that can tell you, for example, if the i-th element of (2) is a repeat or not (assuming you express i in any base, such as binary or decimal, or even if you express i in unary). By contrast, there is an efficient algorithm to do this, which involves taking GCD (greatest common divisors). However, I don't know of a simple algorithm to produce the i-th element of (3). This means that even if you know that a bijection exists, it may be harder to produce an algorithm to describe this bijection.

Example 3.7. Say we have sets of people and movies, respectively

$$P = \{x, y, z\},\ \ M = \{\text{Oppenheimer}, \text{Barbie}, 2001, \text{Encounters}\}$$

(referring to the 2023 movies of *Oppenheimer* and *Barbie*, and to the (older) movies *2001: A Space Odyssey*, and *Encounters at the End of the World*). (Hence $|P| = 3 < 4 = |M|$.) Say that

(1) x has seen Barbie and 2001, but not the other movies;
(2) y has seen Oppenheimer and Barbie, but not the others; and
(3) z has seen Encounters, but not the others.

This gives a map $\text{HasSeen}\colon P \to \text{Power}(M)$, where $\text{HasSeen}(p)$ is the set of movies (in M) seen by p. Hence

$$\begin{aligned}\text{HasSeen}(x) &= \{\text{Barbie}, 2001\},\\ \text{HasSeen}(y) &= \{\text{Oppenheimer}, \text{Barbie}\},\\ \text{HasSeen}(z) &= \{\text{Encounters}\}.\end{aligned}$$

In terms of yes/no tables, the function HasSeen can be represented as:

Is $m \in \text{HasSeen}(p)$?, i.e., has p seen m?	$m =$ Oppenheimer	$m =$ Barbie	$m = 2001$	$m =$ Encounters
$p = x$	no	yes	yes	no
$p = y$	yes	yes	no	no
$p = z$	no	no	no	yes

Of course, since $|P| = 3$ and $|\text{Power}(M)| = 16$, it is clear that the map HasSeen cannot be surjective. Let's use Theorem 3.6 to produce an element of $\text{Power}(M)$ that is not in the image. First, consider we build an injection $h\colon P \to M$ in an arbitrary way; to make matters concrete, let $h\colon P \to M$ be the surjection given by:

$$h(x) = \text{Oppenheimer},\ \ h(y) = \text{Barbie},\ \ h(z) = 2001.$$

The upshot of this choice of h is that we will only care about the following parts of the yes/no table:

Is $m \in \text{HasSeen}(p)$?	$m =$ Oppenheimer	$m =$ Barbie	$m = 2001$	$m =$ Encounters
$p = x$	no			
$p = y$		yes		
$p = z$			no	

Now according to (4) (with $S = P$ and $B = M$),

$$T = \{h(p) \mid p \in P \text{ and } h(p) \notin f(p)\} \tag{6}$$

does not lie in the image of f; since $h(x) = \text{Oppenheimer}$ and $\text{Oppenheimer} \notin f(x)$, T contains x; similarly we see T does not contain Barbie and contains 2001; hence in (6) we have

$$T = \{\text{Oppenheimer}, 2001\},$$

which is therefore not in the image of f. More generally, (5) implies says that as long as T contains Oppenheimer and 2001 and does not contain Barbie, (and

Encounters can lie in T or not), T cannot be in the image of f. Visibly any such T is not in the image of f. The extended yes/no table becomes:

Is $m \in \text{HasSeen}(p)$?	$m =$ Oppenheimer	$m =$ Barbie	$m = 2001$	$m =$ Encounters
$p = x$	`no`			
$p = y$		`yes`		
$p = z$			`no`	
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
Is $m \in T$?	`yes`	`no`	`yes`	Either way

3.4. **Generalized Cantor's Theorem: Surjective Form.**

Theorem 3.8 (Surjective Form of Generalized Cantor's Theorem)**.** *Let $g\colon B \to S$ be a surjection of sets. Then for any map $f\colon S \to \text{Power}(B)$, the image of f is not all of* Power(B)*; specifically the set*

$$T = \Bigl\{ b \in B \mid b \notin f\bigl(g(b)\bigr) \Bigr\} \tag{7}$$

is not in the image of f.

Proof. Assume, to the contrary, that $f(t) = T$ for some $t \in S$. Since g is surjective, there exists a $b \in B$ with $g(b) = t$. Then either $b \in T$ or $b \notin T$; in both cases we easily get a contradiction; we leave the details as an exercise (Exercise 9.2.12). $\square$

Example 3.9. Let P, M and HasSeen: $P \to M$ be as in Example 3.7. Let's use Theorem 3.8 to produce an element of Power(M) that is not in the image. First, we build a surjection $g\colon M \to P$ arbitrarily; so let g be given by

$$g(\text{Oppenheimer}) = x,\ g(\text{Barbie}) = y,\ g(2001) = z,\ g(\text{Encounters}) = z.$$

Then we determine

$$T = \Bigl\{ m \in M \mid m \notin f\bigl(g(m)\bigr) \Bigr\};$$

since g(Oppenheimer) $= x$, and x has not seen "Oppenheimer," we have

$$\text{Oppenheimer} \notin \text{HasSeen(x)} = \text{HasSeen}(g(\text{Oppenheimer})),$$

and so Oppenheimer $\in T$. Similarly we see Barbie $\notin T$, 2001 $\in T$, Encounters $\notin T$. Hence

$$T = \{\text{Oppenheimer}, 2001\};$$

also, visibly T is not in the image of HasSeen. In terms of an "extended yes/no" table we can visualize the above as:

Is $m \in \text{HasSeen}(p)$?	$m =$ Oppenheimer	$m =$ Barbie	$m = 2001$	$m =$ Encounters
$p = x$	`no`			
$p = y$		`yes`		
$p = z$			`no`	`yes`
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
Is $m \in T$?	`yes`	`no`	`yes`	`no`

3.5. **Partial Information Using Generalized Cantor's Theorem.** Another point about our generalized Cantor's Theorems is that they allow you to construct sets that are not in the image of a map $f\colon S \to \mathrm{Power}(B)$ when you only have partial information about f. You can sometimes do this with Cantor's Theorem, but the results are much more limited.

Example 3.10. Say that $S = \{a, b, c\}$ and $f\colon S \to \mathrm{Power}(S)$ is a map such that $a \in f(a)$, $b \in f(b)$, and $c \notin f(c)$. This is only "partial information" on the value of f, but this is enough to determine:

$$T = \{s \in S \mid s \notin f(s)\},$$

namely the set T from Cantor's Theorem 2.8. In particular $T = \{c\}$, since $c \notin f(c)$, and since $s \in f(s)$ for $s = a, b$. Hence this partial information is enough to determine a set, T, that is not in the image of f. The extended yes/no table for this situation is:

Is $y \in f(x)$?	$y = a$	$y = b$	$y = c$
$x = a$	`yes`		
$x = b$		`yes`	
$x = c$			`no`
	$\downarrow$	$\downarrow$	$\downarrow$
Does $x = y$ lie in T?	`no`	`no`	`yes`

Example 3.11. Say that $S = \{a, b, c\}$ and $f\colon S \to \mathrm{Power}(S)$ is a map such that $a \in f(a)$ and $b \in f(b)$, but you know nothing about $f(c)$. This "partial information" is clearly not enough to produce a T that is not in the image of f, since $f(c)$ could be any subset of S. The yes/no table looks like:

Is $y \in f(x)$?	$y = a$	$y = b$	$y = c$
$x = a$	`yes`		
$x = b$		`yes`	
$x = c$			???
	$\downarrow$	$\downarrow$	$\downarrow$
Does $x = y$ lie in T?	`no`	`no`	???

Example 3.12. Say that $S = \{a, b, c\}$ and $f\colon S \to \mathrm{Power}(S)$ is a map such that $a \in f(b)$, $b \in f(c)$, and $c \notin f(a)$. This partial information on f doesn't allow us to determine

$$T = \{s \in S \mid s \notin f(s)\}$$

(indeed, to see whether or not $a \in T$ we need to know if $a \in f(a)$ or $a \notin f(a)$). The yes/no table here looks like:

Is $y \in f(x)$?	$y = a$	$y = b$	$y = c$
$x = a$			`no`
$x = b$	`yes`		
$x = c$		`yes`	

Let's use the Injective Form of Generalized Cantor's Theorem and (4) to find a T not in the image of f: we are given that

$$a \in f(b),\ b \in f(c),\ c \notin f(a);$$

we now search for an injection $h\colon S \to S$ where the above information is equivalent to knowing whether or not $h(x)$ lies in $f(x)$: since the above information is equivalent to

$$c \notin f(a),\ a \in f(b),\ b \in f(c), \tag{8}$$

we set h to be the function

$$h(a) = c,\ h(b) = c,\ h(c) = b \tag{9}$$

so that we have

$$h(a) = c \notin f(a),\ h(b) = a \in f(b),\ h(c) = b \in f(c).$$

Hence we can now determine whether or not $h(x) \in f(x)$ for all $x \in S$; note that luckily h is an injection $S \to S$ (and also a bijection). The yes/no table can be rearranged as:

Is $y = h(x) \in f(x)$?	$y = h(a) = c$	$y = h(b) = a$	$y = h(c) = b$
$x = a$	no		
$x = b$		yes	
$x = c$			yes

Thanks to h we have information on the "diagonal" elements, and we can take:

(10)

Is $y = h(x) \in f(x)$?	$y = h(a) = c$	$y = h(b) = a$	$y = h(c) = b$
$x = a$	no		
$x = b$		yes	
$x = c$			yes
	$\downarrow$	$\downarrow$	$\downarrow$
Does $y = h(x)$ lie in T?	yes	no	no

Hence $h(a) \in T$ and $h(b), h(c) \notin T$, and therefore

$$T = \{h(a)\} = \{c\}.$$

[This table should allow you to directly argue that T is not in the image of f: the first row shows that T cannot equal equal $f(a)$, since $h(a) = c \in T$ but $h(a) = c \notin f(a)$. You should be able to similarly argue that T does not equal $f(b)$ and $f(b)$.] Note that to more easily read off which elements of S lie in T, we can rearrange the columns and rows of (10) as

(11)

Is $y = h(x) \in f(x)$?	$y = h(b) = a$	$y = h(c) = b$	$y = h(a) = c$
$x = b$	yes		
$x = c$		yes	
$x = a$			no
	$\downarrow$	$\downarrow$	$\downarrow$
Does $y = h(x)$ lie in T?	no	no	yes

In the above example, we have a map $S \to \mathrm{Power}(B)$ with $B = S$; ironically, this "simpler" situation is sometimes conceptually more confusing. Here is a similar example with $B \neq S$.

Example 3.13. Say that:

(1) Ursula Le Guin has written *The Dispossessed*, *The Lathe of Heaven*, and *The Left Hand of Darkness*;
(2) Daniel Abraham and Ty Franck have co-written *Leviathan Wakes*.

Let

$$B = \{\text{Dispossessed}, \text{Lathe}, \text{Left}, \text{Leviathan}\}, \quad S = \{\text{Ursula}, \text{Daniel}, \text{Ty}\};$$

the above data gives a function

$$\text{CoWrote}\colon S \to \text{Power}(B)$$

(we say that s cowrote b even if s is the sole author of b); the yes/no table depicting CoWrote is:

Did s cowrite b, i.e., is $b \in \text{CoWrote}(s)$?	$b =$ Dispossessed	$b =$ Lathe	$b =$ Left	$b =$ Leviathan
$s =$ Ursula	yes	yes	yes	no
$s =$ Daniel	no	no	no	yes
$s =$ Ty	no	no	no	yes

Now imagine that we don't know the above, and only know the partial information:

Did s cowrite b, i.e., is $b \in \text{CoWrote}(s)$?	$b =$ Dispossessed	$b =$ Lathe	$b =$ Left	$b =$ Leviathan
$s =$ Ursula			yes	
$s =$ Daniel				yes
$s =$ Ty	no			

We can still produce a subset of novels that is not equal to the set of novels written or co-written by any author; namely, this partial information lets us answer the question "did s cowrite $h(s)$?", or, equivalently, "is $h(s)$ in CoWrote(s)?", provided we set

$$h(\text{Ursula}) = \text{Left}, \ h(\text{Daniel}) = \text{Leviathan}, \ h(\text{Ty}) = \text{Dispossessed};$$

notice that $h\colon S \to B$ is injective. Hence we produce a set of books T not in the image of f via:

Did s cowrite b, i.e., is $b \in \text{CoWrote}(s)$?	$b =$ Dispossessed	$b =$ Lathe	$b =$ Left	$b =$ Leviathan
$s =$ Ursula, $h(s) =$ Left			yes	
$s =$ Daniel, $h(s) =$ Leviathan				yes
$s =$ Ty, $h(s) =$ Dispossessed	no			
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
Does $h(s)$ lie in T?	yes	Either way	no	no

To give this yes/no table a "diagonal look," we rearrange the rows:

Did s cowrite $h(s)$?	$b =$ Dispossessed	$b =$ Lathe	$b =$ Left	$b =$ Leviathan
$s =$ Ty, $h(s) =$ Dispossessed	no			
$s =$ Ursula, $h(s) =$ Left			yes	
$s =$ Daniel, $h(s) =$ Leviathan				yes
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
Does $h(s)$ lie in T?	yes	Either way	no	no

Hence

$$T = \{\text{Dispossessed}\}, \{\text{Dispossessed}, \text{Lathe}\}$$

are two subsets of books that no author cowrote. If we drop the column corresponding to *The Lathe of Heaven*, we get the table

Did s cowrite $h(s)$?	$b =$ Dispossessed	$b =$ Left	$b =$ Leviathan
$s =$ Ty, $h(s) =$ Dispossessed	`no`		
$s =$ Ursula, $h(s) =$ Left		`yes`	
$s =$ Daniel, $h(s) =$ Leviathan			`yes`
	$\downarrow$	$\downarrow$	$\downarrow$
Does $h(s)$ lie in T?	`yes`	`no`	`no`

Notice that in this table, the books in the columns appear in alphabetical order, which is how (11) is organized (i.e., the y's appear in alphabetical order a, b, c), as opposed to how (10) is.

4. Some Unrecognizable Languages

It is important to know that a number of problems are "unsolvable." In this section we make these ideas in rough terms; we will only be precise after we cover *Turing machines* in Chapter 3 of the course textbook [Sip].

4.1. **Alphabets, Strings, and Languages.**

Let us review some common definitions in computer science theory (many can be found in [Sip], Chapter 0).

An *alphabet* is a finite, nonempty set.

Let Σ be an alphabet. For any $k \in \mathbb{Z}_{\geq 0} = \{0, 1, 2, \ldots\}$, the set of *strings of length k over* Σ refers to Σ^k (the Cartesian product $\Sigma \times \ldots \times \Sigma$ of k copies of Σ); hence $\Sigma^0 = \{\epsilon\}$ where ϵ denotes the empty string. The set of *strings over* Σ refers to union

$$\Sigma^* \overset{\text{def}}{=} \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Example 4.1. If $\Sigma = \{a, b\}$, then $(a, b, b, a) \in \Sigma^4$ is a string of length 4 over Σ; for brevity we usually write *abba* instead of (a, b, b, a). Hence

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$$

(we often list elements of a language by shortest first, and secondarily in lexicographical order).

Example 4.2. The set of strings of length two over $\Sigma = \{a, b, c\}$ equals

$$\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\};$$

hence $|\Sigma^2| = 9 = 3^2$ in this example. You should be able to convince yourself that for any alphabet Σ and any $k \in \mathbb{Z}_{\geq 0}$:

(1)
$$\left|\Sigma^k\right| = |\Sigma|^k,$$
and

(2) there is a bijection between: (1) elements of Σ^k and (2) functions $[k] \to \Sigma$, where $[k]$ is shorthand for the set $\{1, 2, \ldots, k\}$ (and $[0] = \emptyset$).

A *language over* Σ refers to any subset of Σ^*. Hence a language is an element of Power(Σ^*), and the set of *all languages over* Σ therefore equals Power(Σ^*).

4.2. **More Languages and Descriptions.**

Example 4.3. Let $\Sigma_{\rm digits} = \{0, 1, \ldots, 9\}$. Every non-negative integer has a unique base 10 representation (e.g., $0, 1, 2, 7, 421, 2023$), which we interpret as a string over $\Sigma_{\rm digits}$. More formally, if $n \in \{0, 1, 2, \ldots\}$, we use $\langle n\rangle$ to denote the associated string; hence $\langle 2024\rangle$ is technically the string of length 4, $(2, 0, 2, 4)$, but for brevity write 2024; to avoid confusion, at times we write "the integer 2024" for the integer, and $\langle 2024\rangle$ or "the string 2024" when referring to the string). We also call $\langle 2024\rangle$ "the description of 2024." We define

$$\text{PRIMES} = \{\langle n\rangle \mid n \text{ is prime}\} = \{\langle 2\rangle, \langle 3\rangle, \langle 5\rangle, \langle 7\rangle, \langle 11\rangle, \langle 13\rangle, \ldots\} \subset \Sigma^*_{\rm digits}$$

or, for brevity,

$$\text{PRIMES} = \{2, 3, 5, 7, 11, 13, \ldots\} \subset \Sigma^*_{\rm digits}.$$

We tend to write languages in ALL CAPITAL LETTERS. When working in mathematics, we usually work with the ("actual") set of prime numbers

$$\{2, 3, 5, 7, 11, 13, \ldots\} \subset \mathbb{N} \subset \mathbb{Z}_{\geq 0} \subset \mathbb{Z}.$$

Example 4.4. When we work with *regular languages* (Chapter 1 of [Sip]), we will discuss languages like:

$$\text{DIV-BY-5} = \{\langle 0\rangle, \langle 5\rangle, \langle 10\rangle, \langle 15\rangle, \ldots\} = \{0, 5, 10, 15, \ldots\} \subset \Sigma^*_{\rm digits}$$

(by writing $\subset \Sigma^*_{\rm digits}$ we know that we are speaking about a subset of strings). Related languages are

$$\text{POSITIVE-DIV-BY-5} = \{5, 10, 15, 20, 25, \ldots\} \subset \Sigma^*_{\rm digits}$$

and

$$\text{DIV-BY-5-LEADING-ZEROS-OK}^{9}$$
$$= \{0, 5, 00, 05, 10, 15, 20, \ldots, 95, 000, 005, 010, \ldots, 095, 100, 105, \ldots\} \subset \Sigma^*_{\rm digits}$$

[When we discuss DFA's in Chapter 1 of [Sip], we[10] may prove that the minimum number of states required to recognize DIV-BY-5-LEADING-ZEROS-OK is 3; by contrast, recognizing either of the other two languages above with a DFA requires more than 3 states; moreover, the language

$$\{\,\epsilon\,\} \cup \text{DIV-BY-5-LEADING-ZEROS-OK}$$

can be recognized by a DFA with 2 states.]

Remark 4.5. Starting in Section 3.3 of [Sip], we will use the notation $\langle X\rangle$ to mean "the description of X as string" (with some agreed upon conventions). For example, if $\mathcal{G}$ is the set of graphs, G, whose vertex set is of the form $[n] = \{1, 2, \ldots, n\}$, then (in class) we will describe each $G \in \mathcal{G}$ as a string $\langle G\rangle$ over some fixed alphabet (e.g., $\Sigma = \{0, 1, \#\}$). Similarly if f is a Boolean formula on variables $x_1, \ldots, x_n$, we will fix some conventions so that $\langle f\rangle$ is a string over some alphabet that describes f (a convenient alphabet here is $\Sigma = \{x, 0, 1, \wedge, \vee, \neg, (,)\}$).

[9] In this language we allow extraneous leading 0's, so, for example, the strings $5, 05, 005 \in \Sigma^*_{\rm digits}$ are identified with the integer 5, which is divisible by 5, and therefore these strings are contained in this language.

[10] Here "we" may actually refer to the students, not the instructor.

Remark 4.6. When we discuss algorithms that run in "polynomial time" in Chapter 7 of [Sip], we will see that there is a big difference between (1) alphabets with at least two symbols and (2) unary alphabets, i.e., those with exactly one symbol; both types of alphabets will be of interest. In this section we discuss only the concepts of *recognizability* and *decidabilty*, where this distinction is unimportant.

4.3. The Language "Duck" (Added Sept 13, 2024).

As a warm-up to the next few subsections, we introduce a toy programming language; we'll call it Duck™[11]. We will describe the details in class; let us give the basic idea.

A Duck program will be a string over Σ_{ASCII}. We define:

(1) a *valid Duck statement* is a string over Σ_{ASCII} which is concatenation $s_1s_2 = s_1 \circ s_2$, where s_1 is the 5 symbol string `quack`, and s_2 is an arbitrary string over $\Sigma_{\mathrm{digits}} = \{0, 1, \ldots, 9\}$[12] For example, the following strings over Σ_{ASCII} are valid Duck statements:

(12) `quack3`, `quack0000`, `quack2024`, `quack421`,

and `quackquack`[13] while the following are not valid Duck statements

(13) `CPSC421`, `Quack2`, `2024`, `woof23`.

(Note: we are using commas (,) and periods (.) here as punctuation marks; the symbols `,` and `.` belong to Σ_{ASCII}, but we'll try to avoid them in examples. For that matter, we'll avoid spaces in our Duck programs.]

(2) Since `quack` $\in \Sigma^*_{\mathrm{ASCII}}$ and $\Sigma_{\mathrm{digits}} \subset \Sigma_{\mathrm{ASCII}}$, we have therefore built a language

$$\text{VALID-DUCK-STATEMENTS} \subset \Sigma^*_{\mathrm{ASCII}}$$

whose elements are all valid duck statements; hence VALID-DUCK-STATEMENTS contains the strings in (12), and not those in (13).

(3) A *valid Duck program* is a finite concatenation of valid Duck statements, such as:

`quack3quack4quack19`, `quack0`.

This gives us a language

$$\text{VALID-DUCK-PROGRAMS} \subset \Sigma^*_{\mathrm{ASCII}}.$$

(4) A *Duck program* is **any** string over the alphabet Σ_{ASCII}.

(5) An *input to a Duck program* is any string over the alphabet Σ_{ASCII}.

(6) Given $p \in$ VALID-DUCK-PROGRAMS and $i \in \Sigma^*_{\mathrm{ASCII}}$, we say that p *accepts* i, if $|i|$, i.e., the length of i, is one of the numbers appearing in the duck program; otherwise we say that p *does not accept* i. Hence the program

$$p = \texttt{quack3quack4quack19}$$

accepts the strings

`aaa`, `abba`, `1234567890123456789`, `quack3quack4quack19`, `UBC`

but does not accept the strings

`aa`, `abbaa`, `12345678901234567`.

[11] Trademark of the 421 Waterfowl Software Foundation.

[12] After some class discussion, we decided to allow s_2 to be the empty string; see Remark 4.7.

[13] See Remark 4.7.

(7) For each $p \in$ VALID-DUCK-PROGRAMS, we define the language recognized by p to be the language

$$\mathrm{LanguageRecBy}(p) \overset{\mathrm{def}}{=} \{i \in \Sigma^*_{\mathrm{ASCII}} \mid p \text{ accepts } i\}.$$

If p is an ASCII string not in VALID-DUCK-PROGRAMS, for any $i \in \Sigma^*_{\mathrm{ASCII}}$ we say that p *does not accept* i, and we define

$$\mathrm{LanguageRecBy}(p) \overset{\mathrm{def}}{=} \emptyset.$$

(8) Hence we have defined a map

$$\mathrm{LanguageRecBy}\colon \Sigma^*_{\mathrm{ASCII}} \to \mathrm{Power}(\Sigma^*_{\mathrm{ASCII}}) \tag{14}$$

(9) We say that a language $L \subset \Sigma^*_{\mathrm{ASCII}}$ is *recognizable* if it is in the image of LanguageRecBy; otherwise we say that it is *unrecognizable*.

This gives you the rough idea of what is meant by "p accepts i" and "p does not accept i" in the context of the Duck programming language.

Remark 4.7. In class on September 13, 2024, we found that the above description of Duck was not completely precise. For example, it is unclear what `quack0003` will accept; after a board meeting, we decided to allow leading zeros. Hence

$$\mathrm{LanguageRecBy}(\texttt{quack0003}) = \mathrm{LanguageRecBy}(\texttt{quack3}) = \Sigma^3_{\mathrm{ASCII}}.$$

Also, if ϵ is the empty string in Σ_{ASCII}, then $\texttt{quack}\epsilon = \texttt{quack}$ is a valid Duck statement, and hence `quackquack5` is a valid Duck program. After a contentious board meeting and coin flip we decided that we ignore any appearance of $\texttt{quack}\epsilon = \texttt{quack}$, and therefore

$$\mathrm{LanguageRecBy}(\texttt{quackquack5}) = \Sigma^5_{\mathrm{ASCII}}.$$

Finally, we decided that the empty string is *not* a valid Duck program, and hence using the common notation:

$$L^* = \bigcup_{k \in \mathbb{Z}_{\geq 0}} L^k = L^0 \cup L^1 \cup L^2 \cup \dots$$

$$L^+ = \bigcup_{k \in \mathbb{N}} L^k = L^1 \cup L^2 \cup L^3 \cup \dots = LL^*$$

(and powers are taken to mean concatenation of languages, e.g., $L^2 = L \circ L$ rather than the Cartesian product $L \times L$), we have

$$\text{VALID-DUCK-PROGRAMS} = \big(\text{VALID-DUCK-STATEMENTS}\big)^+.$$

It should be clear that $\{\texttt{a}\}$ is not recognizable (in the context of Duck programs). The reason is that if a Duck program, p, accepts the string `a`, then p must accept all strings of length 1. Moreover if p is not a valid duck program, then by definition $\mathrm{LanguageRecBy}(p) = \emptyset$.

For the same reason we have $L \subset \Sigma^*_{\mathrm{ASCII}}$ is recognizable (in the context of Duck programs) iff there exists an $m \in \mathbb{Z}_{\geq 0}$ and $k_1, \dots, k_m \in \mathbb{Z}_{\geq 0}$ such that

$$L = \Sigma^{k_1}_{\mathrm{ASCII}} \cup \dots \Sigma^{k_m}_{\mathrm{ASCII}}.$$

(the case $m = 0$ and therefore $L = \emptyset$[14] is also recognizable, namely by any string that is not a valid Duck program).

[14]The empty set is the "union of 0 sets" or the "empty union." To understand why, consult your local expert on the empty set.

Hence there is a simple way to classify all languages that are Duck-recognizable. Cantor's theorem tells us that

$$T = \{p \in \Sigma_{\mathrm{ASCII}}^* \mid p \notin \mathrm{LanguageRecBy}_{\mathrm{Duck}}(p)\}$$

is not Duck-recognizable. (At this point we will add the prefix "Duck-" to a lot of terms, to avoid confusion with other notions of recognizable.) Therefore T contains the strings

`QUACK`, `woofwoof`, `abba`, `quack3`, `quackquack20`

(and contains every string that is not a valid Duck program, including ϵ), but T does not contain the strings

`quack6`, `quack07`, `quack3quack4quack19`, `quackquack12`.

This subsection should provide some intuitive basis for the use of Cantor's theorem in the next few subsections, where we discuss the more common notion of a *recognizable* versus an *unrecognizable* language.

4.4. The Language Recognized by a Valid Python Program.

In this section we will provide a rough framework needed to define "recognizable" languages in the usual sense. You should be convinced that this framework can be made precise (there are choices to make), and that the notion of a "recognizable language" does not depend on any the choices you make. The upshot is that we:

(1) define Σ_{ASCII} to be the usual ASCII alphabet;
(2) define a subset VALID-PYTHON-PROGRAMS of $\Sigma_{\mathrm{ASCII}}^*$ (hence VALID-PYTHON-PROGRAMS is a language over Σ_{ASCII});
(3) for each $p \in$ VALID-PYTHON-PROGRAMS, we define the (language of) *strings over* Σ_{ASCII} *accepted by* p;
(4) and we define the function

$$\mathrm{LanguageRecBy} \colon \Sigma_{\mathrm{ASCII}}^* \to \mathrm{Power}(\Sigma_{\mathrm{ASCII}}^*) \tag{15}$$

given by $\mathrm{LanguageRecBy}(p) = \emptyset$ if p is not a valid Python program, and otherwise $\mathrm{LanguageRecBy}(p)$ is the set of strings that p accepts.
(5) If $L \subset \Sigma_{\mathrm{ASCII}}^*$, we say that L is *recognizable* if it is in the image of LanguageRecBy, i.e., if there is a valid Python program, p, that recognizes L, i.e., if L is the set of strings accepted by p.

Everything we say in this subsection and Subsection 4.5 is valid in the context of any function LanguageRecBy **in** (15)**. Hence the "fine print" below is designed to indicate what terms like "recognizing" and "accepting" will mean in Chapter 3 of [Sip], using *Turing machines*, which is a simple (but limited) model of a "computer program."**

4.4.1. *The Fine Print.*

Let Σ_{ASCII} be the usual ASCII alphabet, an alphabet of size 256. Fix a programming language, such as Python (almost any other programming language will work, e.g., C, C++, APL, Javascript, MATLAB, Maple, etc.).

To fix ideas, here is a sample Python program (called a "function" or "user defined function" in Python), named `isPal()`, that checks if an "input" to the program is a palindrome (i.e., the same as the reverse word):

```
#
# Python ignores any part of a line after the first "#"
#
def isPal():                                # the program is called "isPal()"
```

```
    i = input("Your input: ")            # i is an input given by the user
    n = len( i )                         # n is the length of i
    for m in range( n ):                 # hence m runs from 0 to n-1
        if ( i[m] != i[ n-1-m] ):        # Note that != means "not equal to"
            return("no")                 # no,  i  is not a palindrome
    return("yes")                        # yes,  i  is a palindrome
```

[This program takes an ASCII string `i` as input, sets `n` to be the length of `i`, and then checks if the m-th character of i equals the $(n-m-1)$-th character of i, for $m = 0, 1, \ldots, n-1$.]

N.B.: to get the above function to run in Python, you can add a line:

```
print( "\n   Is the input a palindrome? " + isPal() + "\n" )
```

after the function, give the file a name, and run `python` on this file.

Assume you have fixed conventions so that the following holds (there is some flexibility here, but most "reasonable"[15] set of conventions will work): there is a subset

$$\text{VALID-PYTHON-PROGRAMS} \subset \Sigma^*_{\text{ASCII}}$$

(i.e., a language over Σ_{ASCII}) that are designated to be "valid Python programs" such that if p is a valid Python program, then, after discarding the comments (i.e., the `#`'s and anything that follows them on a line)

(1) The first line of p—although irrelevant to us—is a "`def` statement" that names the program and declares it to have no arguments (as in the above example).
(2) The second line of p is `i = input("Your input:  ")`, that sets the variable `i` to an arbitrary ASCII string specified "externally" by the "user;" we refer to this value of `i` as the *input* to the program.
(3) No other lines of the program use an `input` statement.
(4) After the line `i = input("Your input:  ")`, the program runs with the "usual conventions" of Python (here there is some flexibility).
(5) When a `return` statement is reached, the program execution stops. The program may stop for other reasons (e.g., if you divide by 0, or if it runs the entire program). A program may never stop, for various reason (e.g., if it tries to execute something that is obviously an "infinite loop," if it tries to find an even prime number greater than 2, maybe if it searches for an odd perfect number[16], etc.).
(6) (Looking forward a bit:) We say that p *accepts* i (respectively, p *rejects* i) if p is a valid Python program that on input i eventually reaches a `return("yes")` statement (respectively, a `return("no")` statement. If p is a valid Python program that does not accept or reject i, we say that p *loops on* i.

[It should be pretty clear how to make the above precise, at least to readers who have had the pleasure of writing computer programs. As mentioned, there is some flexibility here; in class I'll answer questions you have regarding the above. When we cover *Turing machines* we will have precise definitions of (1) a "program" (or *Turing machine*, (2) an "input" to a program, and (3) when a program *accepts* or *rejects* a program.]

[15] In class we may give examples of "unreasonable" conventions, such as that in Example 4.34.

[16] Whether or not there exists an odd perfect number is still, I believe, an open problem.

Remark 4.8. From the above, it follows that p can *loop* on an input, i, for various reasons: it can reach a `return` statement that does not return `"yes"` or `"no"`; it can stop (e.g., by reaching its end without encountering a `return` statement, by dividing by zero, etc.); or it can never stop.

By a *pseudo-definition* we mean a definition that is not precise, but that you can probably make precise in some reasonable way (if you read the "fine print," Subsubsection 4.4.1).

Pseudo-Definition 4.9. Say that $p \in \Sigma^*_{\rm ASCII}$ is a valid Python program, and $i \in \Sigma^*_{\rm ASCII}$. We say that p *accepts* i if, when p is run on input i, p eventually reaches the statement `return("yes")`. The *language recognized by* p is defined to be

$$\text{LanguageRecBy}(p) \overset{\text{def}}{=} \{i \in \Sigma^*_{\rm ASCII} \mid p \text{ accepts } i\} \subset \Sigma^*_{\rm ASCII}. \tag{16}$$

If p is not a valid Python program, we define

$$\text{LanguageRecBy}(p) \overset{\text{def}}{=} \emptyset$$

(one could replace $\emptyset$ by any language recognized by a valid Python program). (Hence LanguageRecBy is a map $\Sigma^*_{\rm ASCII} \to \text{Power}(\Sigma^*_{\rm ASCII})$.) We say that $L \subset \Sigma^*_{\rm ASCII}$ is *recognizable* if there is a valid Python program p such that $L = \text{LanguageRecBy}(p)$, and otherwise we say that L is *unrecognizable*.

Example 4.10. The program `isPal()` above recognizes the language PALINDROME, of ASCII strings that are palindromes. Hence PALINDROME is recognizable.

Example 4.11. The language PRIMES is recognized by a Python program (assuming a typical interpretation of Python programs); indeed, the program needs to (1) checks if the input is string representing an integer greater than 1, and (2) check if the integer has a divisor other than itself and 1. [The naive version of checking (2) takes "exponential time" in the length of the input.] Similarly for DIV-BY-5 and many other languages (CPSC 320 gives many examples).

4.4.2. *If Didn't Read the Fine Print Above.* Let LanguageRecBy be an arbitrary function as in (15), i.e., $\text{LanguageRecBy}\colon \Sigma_{\rm ASCII} \to \Sigma^*_{\rm ASCII}$. From (15) alone, we can define all the terms we need in Subsection 4.5:

(1) if $p, i \in \Sigma^*_{\rm ASCII}$, we say that p *accepts* i if $i \in \text{LanguageRecBy}(p)$;
(2) if $p \in \Sigma^*_{\rm ASCII}$, we refer to $\text{LanguageRecBy}(p)$ as the language recognized by p;
(3) we say that $L \subset \Sigma^*_{\rm ASCII}$ is *recognizable* if it is in the image of LanguageRecBy; otherwise L is *unrecognizable*.

Nonetheless, for Subsection 4.6 you will need to be (at least vaguely) aware of the fine print above.

4.5. **An Unrecognizable Language.**

Theorem 4.12. *Let* LanguageRecBy *be any function* (15). *Then the language*

$$\bigl\{p \in \Sigma^*_{\rm ASCII} \mid p \notin \text{LanguageRecBy}(p)\bigr\}$$

is unrecognizable.

Proof. This follows immediately from Theorem 2.8, taking $S = \Sigma^*_{\rm ASCII}$ and $f =$ LanguageRecBy there (the rest is "unwinding" the definitions). □

Remark 4.13. In class on September 13, 2023, we named the language in the above theorem

$$\text{GROUCHO-MARX-SELF} = \{p \in \Sigma^*_{\rm ASCII} \mid p \notin \text{LanguageRecBy}(p)\}$$

based on the quote:

> "I don't want to belong to any club that will accept me as a member." Groucho Marx (1890–1977).

We also thought of other names for this language. However, it might be truer to the quote to define

$$\text{GROUCHO-MARX-WANTS-IN} = \{p \in \Sigma^*_{\rm ASCII} \mid \text{GROUCHO-MARX} \notin \text{LanguageRecBy}(p)\}.$$

It's unclear if we will use this terminology after 2023...

Remark 4.14. Note that by our conventions,

$$T = \bigl\{p \in \Sigma^*_{\rm ASCII} \mid p \notin \text{LanguageRecBy}(p)\bigr\}$$

includes all strings, p, that are not valid Python programs. Say that our conventions regarding valid Python programs imply that there is some symbol $\sigma_0 \in \Sigma_{\rm ASCII}$ that is never found in a valid Python program; it follows that for $k \in \mathbb{N}$ sufficiently large, 99.9999% of the ASCII strings of length k do not represent valid Python programs (why?). It follows that T and the language $\Sigma^*_{\rm ASCII}$ agree on 99.9999% of the ASCII strings of any sufficiently large length. Since $\Sigma^*_{\rm ASCII}$ is recognizable (by a program that always returns `"yes"`), T can "mostly agree" with a recognizable language. Theorem 4.12 asserts only that no algorithm can correctly recognize T "100% of the time."

4.6. Undecidable and Unrecognizable Languages.

There is no obvious reason why you'd want to produce an algorithm to recognize

$$T = \{p \in \Sigma^*_{\rm ASCII} \mid p \notin \text{LanguageRecBy}(p)\}. \tag{17}$$

However, the consequences of the unrecognizability of T are rather drastic.

Here it will be important to roughly understand the "fine print" above (Subsubsection 4.4.1).

Assume some conventions regarding Python programs are fixed as in Subsection 4.4.

Convention 4.15. Let σ_0 be ASCII symbol 28 (from 0–127), i.e., 1C (in Octal $\times$ Hexidecimal), i.e., the $\langle\text{FS}\rangle$ (File Separator) symbol.[17] We fix the convention that no valid Python program can contain the ASCII symbol σ_0.

The following proposition is almost immediate, but is worth stating formally.

Proposition 4.16. *Let $s \in \Sigma^*_{\rm ASCII}$. Then:*

(1) if s contains the symbol $\sigma_0 = \langle\text{FS}\rangle$, then there is a unique way to write s as $p\sigma_0 i$ where $p \in \Sigma^*_{\rm ASCII}$ does not contain σ_0 and $i \in \Sigma^*_{\rm ASCII}$;

[17] The ASCII character set includes some symbols that are designed to be "separators," namely FS,RS,GS,US (file-, record-, group-, and unit-separator), characters 1C–1F, i.e., 28–31. Any of those would do.

(2) if s does not contain the symbol $\sigma_0 = \langle \mathrm{FS} \rangle$, then there is no way to write s as $p\sigma_0 i$ for any $p, i \in \Sigma^*_{\mathrm{ASCII}}$.

[To prove this, note that if s contains symbol σ_0, then p must be the word before the first σ_0. We leave a formal proof to the interested reader(s).]

Hence if p is any valid Python program, and i an input to p, we can encode the pair (p, i) as the string $s = p\sigma_0 i$; given such an s, we can recover p as the substring of s occurring before the first σ_0 in s, and recover i as the substring after the first σ_0.

Notation 4.17. We use the notation:

$$\begin{aligned} \text{NOT-PYTH-INP} &= \{s \in \Sigma^*_{\mathrm{ASCII}} \mid s \text{ cannot be written as } p\sigma_0 i \text{ where } p \text{ is a valid Python program}\}, \\ \text{ACCEPTANCE} &= \{s \in \Sigma^*_{\mathrm{ASCII}} \mid s = p\sigma_0 i \text{ where } p \text{ is a valid Python program that accepts } i\}, \\ \text{REJECTION} &= \{s \in \Sigma^*_{\mathrm{ASCII}} \mid s = p\sigma_0 i \text{ where } p \text{ is a valid Python program that rejects } i\}, \\ \text{LOOPING} &= \{s \in \Sigma^*_{\mathrm{ASCII}} \mid s = p\sigma_0 i \text{ where } p \text{ is a valid Python program that loops on } i\}. \end{aligned}$$

The following propositions are almost immediate, but they are worth stating formally.

Proposition 4.18. *Every string in $\Sigma^*_{\mathrm{ASCII}}$ is in exactly one of the four languages in Notation 4.17.*

The proof merely "unwinds the definitions."

We can restate Proposition 4.18 as saying that $\Sigma^*_{\mathrm{ASCII}}$ is *partitioned* into the four subsets in Notatoin 4.17.

Proposition 4.19. *Let*

$$\begin{aligned} T &= \bigl\{q \in \Sigma^*_{\mathrm{ASCII}} \bigm| q \notin \text{LanguageRecBy}_{\text{Python}}(q)\bigr\} \\ &= \bigl\{q \in \Sigma^*_{\mathrm{ASCII}} \bigm| q \text{ does not accept (the input) } q\bigr\}, \end{aligned} \tag{18}$$

and let $q \in \Sigma^*_{\mathrm{ASCII}}$. Then

$$\begin{aligned} q\sigma_0 q \in \text{NON-PYTH-INP} &\Rightarrow q \in T, \\ q\sigma_0 q \in \text{ACCEPTANCE} &\Rightarrow q \notin T. \\ q\sigma_0 q \in \text{REJECTION} &\Rightarrow q \in T. \\ q\sigma_0 q \in \text{LOOPING} &\Rightarrow q \in T. \end{aligned}$$

Proof. The proof is a straightfoward case analysis. If $q\sigma_0 q \in$ NOT-PYTH-INP then q is not a valid python program; in this case LanguageRecBy$(q) = \emptyset$ (by convention) and so $q \notin$ LanguageRecBy(q) and so $q \in T$. Otherwise $q\sigma_0 q$ lies in (exactly) one of ACCEPTANCE, REJECTION, LOOPING, and in these cases q respectively accepts, rejects, or loops on q (as input), and so, respectively, $q \notin T$, $q \in T$, and $q \in T$. □

Lemma 4.20. *There is a Python algorithm (i.e., program) $u \in \Sigma^*_{\mathrm{ASCII}}$, such that for each string $s \in \Sigma^*_{\mathrm{ASCII}}$:*

(1) If $s \in$ NOT-PYTH-INP, then u terminates on input s and returns the string `"not valid"`.

(2) If $s \in$ ACCEPTANCE, then u terminates on input s and returns the string `"accepted"`.

(3) *If $s \in$ REJECTION, then u terminates on input s and returns the string* `"rejected"`.
(4) *If $s \in$ LOOPING, then u may or may not terminate on input s; if it terminates, the algorithm returns the string* `"loops"`.

NOTE: we use the word "terminate" to mean that the Python program, run in the way a Python program is supposed to be run, reaches a `return` statement or executes the last line of the program (and has "nothing left to do"), or possibly gives an error message when you execute a line of code[18]. Hence if a Python program on a given input does not "terminate," then it "loops" because it doesn't accept or reject the input.

Proof. (This is not a detailed proof.) The algorithm begins by checking whether or not $s \in$ NOT-PYTH-INP: this can be done by checking whether or not σ_0 occurs somewhere in s, and, if so, writing $s = p\sigma_0 i$ (see Proposition 4.16). It then checks whether or not p is a valid Python program; here we "wave our hands" and claim this can be done, although likely you'll have to build a "parser," and this parser depends on your Python conventions. [Note: this "parsing" is far easier to do for *Turing machines* as opposed to Python programs, since Turing machines are far simpler (and more limited) in the way they run.]

If $s = p\sigma_0 i$ where $p \in$ VALID-PYTHON-PROGRAMS, then our algorithm will work like a "debugger" that "simulates" what happens when the Python program p is run with input i. Again you should be able to convince yourself that such a "debugger" or "simulator" can be built (in Python). This is far more tedious to build than a mere Python parser; indeed, you will first have to parse p to figure out how it is supposed to be run. □

Definition 4.21. Any $u \in \Sigma^*_{\mathrm{ASCII}}$ that satisfies (1)–(4) of Lemma 4.20 is called a *universal Python program*. [It does not have to work the way the algorithm given in the proof works.]

Corollary 4.22. *Let $s \in \Sigma^*_{\mathrm{ASCII}}$. Then if $s \notin$ LOOPING, then any universal Python program halts on input s. If a universal Python program halts on s, then the string it returns correctly identifies the language of Notation 4.17 in which it lies.*

[By contrast, there is no "universal Duck program," i.e., a program — written in Duck (not, say, in `awk` or Python) — that can "simulate" the result of a Duck program on an input to this program. see Exercise 9.2.35.]

Remark 4.23. Python contains a universal Python function of sorts, namely `exec()`. For example, if in the Python interpreter you set the string `code` via:

```
code = """
i = input("Your input: ")
n = int(i)
if ( n % 2 == 0):
  print("yes, your input is even")
```

[18]This depends on how you define the particular characteristics of a Python program. So if you want to allow the use of the Python function `int()`, which takes a string and returns an integer, then the line `int("asdf")` may cause your Python program to report an error message and to terminate.

```
else:
  print("no, your input is not even")
"""
```

(you need to enter a blank line after the `"""`), then every time you type the command

```
exec(code)
```

you will be asked to input an integer, and will be told if the integer is even or odd.

Notation 4.24. We use the notation:

$$\text{HALTING} = \text{ACCEPTANCE} \cup \text{REJECTION}.$$

The complements of ACCEPTANCE and HALTING are: In addition, we set

$$\begin{aligned} \text{ACCEPTANCE}^{\text{Comp}} &= \Sigma^*_{\text{ASCII}} \setminus \text{ACCEPTANCE} \\ &= \text{NOT-PYTH-INP} \cup \text{REJECTION} \cup \text{LOOPING}, \\ \text{HALTING}^{\text{Comp}} &= \Sigma^*_{\text{ASCII}} \setminus \text{HALTING} \\ &= \text{NOT-PYTH-INP} \cup \text{LOOPING}. \end{aligned}$$

These differ from

$$\begin{aligned} \text{NON-ACCEPTANCE} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that does not accept } i\} \\ &= \text{REJECTION} \cup \text{LOOPING}, \\ \text{NON-HALTING} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that does not halt on } i\} \\ &= \text{LOOPING} \end{aligned}$$

Theorem 4.25. *The languages NON-ACCEPTANCE and NON-HALTING are (Python-)unrecognizable.*

Proof. By Cantor's theorem, there is no Python program that recognizes

$$T = \bigl\{q \in \Sigma^*_{\text{ASCII}} \;\bigm|\; q \notin \text{LanguageRecBy}_{\text{Python}}(q)\bigr\}$$

For the sake of contradiction, say that NON-ACCEPTANCE is recognizable; let us give an algorithm recognizing T (which is impossible). Consider the Python program that on input $q \in \Sigma^*_{\text{ASCII}}$ does the following: in parallel, it performs one step of Algorithm 1, then one step of Algorithm 2, then two steps of Algorithms 1 and 2, then three steps of Algorithms 1 and 2, etc., where:

(1) Algorithm 1 runs a universal Python program, u, on $q\sigma_0 q$; u returns the result `accepted`, then Algorithm 1 declares that $q \notin T$; if u returns any other result, then Algorithm 1 declares that $q \in T$. (Algorithm 1 may never stop running if $q\sigma_0 q \in \text{LOOPING}$.)
(2) Algorithm 2 runs an algorithm, r, for NON-ACCEPTANCE on input $q\sigma_0 q$; and if $q\sigma_0 q$ is accepted by r, then Algorithm 2 declares $q \in T$. If r rejects or loops on $q\sigma_0 q$, then Algorithm 2 enters an infinite loop (if r stops running).

In view of Proposition 4.19 and Definition 4.21, when either Algorithms 1 or 2 stops running, they correctly identify whether $q \in T$ or $q \notin T$; by Definition 4.21, Algorithm 1 stops running whenever

$$q\sigma_0 q \in \bigl(\text{NOT-PYTH-INP} \cup \text{ACCEPTANCE} \cup \text{REJECTION}\bigr),$$

and Algorithm 2 stops running whenever

$$q\sigma_0 q \in \text{LOOPING}.$$

Hence this hybrid algorithm always stops and identifies whether or not $q \in T$ or $q \notin T$, which is impossible.

Now notice that we may similarly recognize T if we replace r used in Algorithm 2 by any r such that $L = \text{LanguageRecBy}(r)$ satisfies

$$\text{LOOPING} \subset L \subset \bigl(\text{NON-PYTH-INP} \cup \text{REJECTION} \cup \text{LOOPING}\bigr) \tag{19}$$

$$= \text{ACCEPTANCE}^{\text{Comp}}.$$

Hence any such L is unrecognizable, including NON-HALTING. □

Scholium 4.26. *Let L be any language satisfying (19). Then L is not (Python-)recognizable.*

Remark 4.27. The reason that Algorithm 2 enters an infinite loop when r rejects $q\sigma_0 q$ is simply to let Algorithm 1 take over to determine if $q \in T$ or $q \notin T$; one could equivalently have Algorithm 2 stop and return some message like `"I don't know, but Algorithm 1 can tell us"` (telling us to stop with Algorithm 2 at that point and to only run Algorithm 1).

Definition 4.28. We say that a Python program is a *decider* if it halts (terminates) in a finite number of execution steps on any input and returns either `yes` or `no`; we say that a language is *decidable* if it is recognized by some Python program that is a decider.

Remark 4.29. Let us summarize these definitions, plus some extra ones given in class on September 13, 2023 (and likely this year).

Term	**Definition**
p *accepts* i	on input i, p returns `yes`
p *rejects* i	on input i, p returns `no`
p *loops* on i	on input i, p does not return `yes` or `no`
p *halts* on i	on input i, p returns `yes` or `no`, i.e., p accepts or rejects i, i.e., p does not loop on i
$\text{LanguageRecBy}(p)$	$\{i \in \Sigma^*_{\text{ASCII}} \mid p \text{ accepts } i\}$
an $L \subset \Sigma^*_{\text{ASCII}}$ is *recognizable*	for some p, $L = \text{LanguageRecBy}(p)$
p is a *decider*	on all inputs, i, p halts on i
an $L \subset \Sigma^*_{\text{ASCII}}$ is *decidable*	for some decider p, we have $L = \text{LanguageRecBy}(p)$

Here we list a few easy facts; we will likely discuss why they are true in class.

Proposition 4.30. *If L is decidable, then also $L^{\text{Comp}} = \Sigma^*_{\text{ASCII}} \setminus L$ is decidable. If L is decidable, then L is recognizable. If L and $L^{\text{Comp}} = \Sigma^*_{\text{ASCII}} \setminus L$ are both recognizable, then they are both decidable. In particular*

$$L \text{ is decidable} \iff L \text{ and } L^{\text{Comp}} \text{ are recognizable,} \tag{20}$$

and

$$L \text{ is unrecognizable} \Rightarrow L \text{ and } L^{\text{Comp}} \text{ are undecidable.} \tag{21}$$

Example 4.31. ACCEPTANCE is undecidable by the proposition above, since $\text{ACCEPTANCE}^{\text{Comp}}$ is unrecognizable by Scholium 4.26. Similarly for REJECTION and HALTING. Using a universal Turing machine, it is easy to show that ACCEPTANCE, REJECTION, and HALTING are all recognizable.

Hence the halting problem $\text{HALTING} = \text{HALTING}_{\texttt{Python}}$ is recognizable but undecidable.

Note that aside from (20) and (21), we will also use "reductions" to produce undecidable and unrecognizable languages.

In class in 2024, we covered this example in class.

Example 4.32. Let

$$\begin{aligned} L &= \{p \mid p \text{ accepts at least one of its possible inputs}\} \\ &= \{p \mid \exists i \in \Sigma^*_{\text{ASCII}} \text{ s.t. } p \text{ accepts } i\} \end{aligned}$$

We can recognize L by listing the elements of Σ^*_{ASCII} as $i_1, i_2, i_3, \ldots$ (since Σ^*_{ASCII} is countable). To recognize L, on input p we check if p is a valid Python program; if so, we run (i.e., simulate) p for one execution step on i_1; then we run p for two execution steps on i_1 and on i_2; then we run p for three execution steps on i_1, i_2, and on i_3; etc. (In class 2024, we described a slightly different way.) If p accepts in any phase of this algorithm, we return `"yes"` (i.e., yes, $p \in L$) It follows that this procedure accepts p iff $p \in L$. Hence L is recognizable. (We will use this type of argument a lot...) Next we use a reduction to show that L is undecidable: so say that L is decidable; we will use a decider for L to show that ACCEPTANCE is decidable: given $p\sigma_0 j$, we form a Python program q that takes p, and adds the line `i = j` after the input statement `i = input("Your input:  ")` (e.g., on the string $p\sigma_0 37$, q is the same as p except we add the line `i = 37` right after the line in p that reads the input). Hence q essentially ignores its input, and uses the value of j as its input. We easily see that $p\sigma_0 j \in \text{ACCEPTANCE}$ iff q accepts any input, iff q accepts at least one input, iff $q \in L$. If follows that L is undecidable but recognizable.

Example 4.33. Let USED-LINE-OF-CODE be the set of strings of the form $p\sigma_0\ell$, where p is a valid Python program and ℓ is a line of the program p that is "executed/reached" when p runs on at least one of its inputs. Then USED-LINE-OF-CODE is undecidable, given that ACCEPTANCE is undecidable, for the following reason (we'll get used to this type of argument): given a program p and an input, i to p, one can produce a program p' with a line of code, ℓ, such that p accepts i iff p' executes ℓ on all inputs to p', by a standard type of construction[19]; hence if USED-LINE-OF-CODE were decidable, then so would be ACCEPTANCE. Moreover, we will also prove that USED-LINE-OF-CODE is recognizable[20]. Hence

[19]To see this, you take p' to be a program that ignores its input and instead sets `SpecialInput` to i; then p' runs like p, except that you create a new line, ℓ, in p', where p' sets `SpecialOutput` to `yes`, and you require that wherever p assigns a value to `SpecialOutput`, you first check if this value is `yes`, and if so then you branch to ℓ. Hence $p'\sigma_0\ell$ lies in USED-LINE-OF-CODE iff $p\sigma_0 i$ lies in ACCEPTANCE.

[20]This is another type of argument that we will get used to: first, one can write the set of possible inputs to a program in an (infinite) list $i_1, i_2, i_3, \ldots$, where (1) $i_1 = \epsilon$, (2) $i_2, \ldots, i_{257}$ are the elements of Σ^1_{ASCII}, (3) $i_{258}, \ldots, i_{1+256+256^2}$ are the elements of Σ^2_{ASCII}, etc. To recognize USED-LINE-OF-CODE, on input $p\sigma_0\ell$, we check if p is a valid Python program; if so, we run

the complement of USED-LINE-OF-CODE (and therefore the analogously defined UNUSED-LINE-OF-CODE) is unrecognizable.

4.7. *Some Subtle Issues.

Example 4.34. Say that PythonReal is a language that is based on Python, but is also allowed access to a real constant, $x \in \mathbb{R}$ as part of its program description; also assume that the operations $+, -, \times$ are performed exactly, as well as the logical operator >= (greater than or equal to). Then if Σ is a fixed alphabet, then it is not hard to build for each language over Σ a PythonReal program that recognizes it. [To do so, set up a bijection Σ^* with $\mathbb{N}$, and to any subset $L \subset \mathrm{Power}(\mathbb{N})$ let

$$x = \sum_{n \in L} 3^{-n},$$

which is a real number between 0 and 1/2; note that $1 \in L$ iff $3x \geq 1$. The rest is an exercise.]

Example 4.35. A similar comment holds for a Python program that is allowed to access an infinitely long string, x, as part of its program description.

In the above two examples, as x varies you are allowing for an uncountable number of programs; these two examples would be considered "unreasonable" as models for algorithms that involve languages over a finite alphabet. However, a PythonReal program—without an arbitrary hardwired constant—is an interesting model for problems involving real computation[21]: surely a lot of numerical algorithms are most directly explained as real number computations; of course, working with finite precision (or exact arithmetic with rational numbers) can introduce additional hurdles when modeling computation and solving problems.

The following will be explained when we cover Chapter 3 and/or a part of Chapter 9 of [Sip].

Example 4.36. A similar comment holds for *oracle Turing machines* with an oracle $A \subset \Sigma^*$ for some alphabet Σ. However, in this context we usually fix the same oracle, A, to be used by all machines. Also, the term *oracle* clarifies that an $A \subset \Sigma^*$ is a part of the machine. (And it is immediate that a language, A, can be decided—in constant time—by a Turing machine using a single oracle call to the oracle A.)

5. Some "Paradoxes"

Two important results of CPSC 421/501 are: (1) the unsolvabilitiy of the halting problem, and (2) NP-completeness. The first is linked with a number of other remarkable results in logic and computing, and appear as paradoxes:

(1) I am lying.
(2) This statement is a lie.

(i.e., simulate) p for one execution step on i_1; then we run p for two execution steps on i_1 and on i_2; then we run p for three execution steps on i_1, i_2, and on i_3; etc. We halt this procedure and accept p if we reach line ℓ on any of these runs (i.e., simulations). It follows that this procedure accepts p iff there is some input, i, such that p reaches line ℓ on input i after some number of steps. Hence USED-LINE-OF-CODE is recognizable.

[21]See, e.g., *Complexity and Real Computation*, by Blum, Cucker, Shub, and Smale, 1998, Springer.

(3) The phrase: "the smallest positive integer not defined by an English phrase of fifty words or fewer" [This is called the "Berry Paradox," although likely due to Russell.]
(4) This is a statement that does not have a proof that it is true.
(5) Leslie writes about (and only about) all those who don't write about themselves.
(6) Let S be "the set of all sets that do not contain themselves." [This is Russell's most famous (and serious) paradox.]
(7) Consider a C program, P, that (1) takes as input a string, i, (2) figures out if i is the description of a C program that halts on input i, and (3) (i) if so, P enters an infinite loop, and (ii) otherwise P stops running (i.e., halts). [The paradox is: what happens when this program is given input j where j is the string representing P ?]

One thing that these statements have in common is that they all either explicity "refer to themselves" or can be "applied to themselves." Another is that they involve fundamental ideas in logic or computing. Another is that on some naive level they lead to a "paradox."

Consider the first statement, "I am lying," famously used, of course, by Captain Kirk and Mr. Spock[22] to destroy the leader of a group of robots. This leads to a paradox: if the speaker is telling the truth, then the speaker is lying ("I am lying"), but if the speaker is lying, then the speaker is lying that they are lying, meaning they are telling the truth. Either way we get a contradition.

All the other statements lead to "paradoxes" (of somewhat different types); this will be discussed in class and the exercises.

Note the similarity with the proof of Cantor's Theorem 2.8, that takes a map $f\colon S \to \mathrm{Power}(S)$ and constructs the set

$$T = \{s \in S \mid s \notin f(s)\}.$$

There is no paradox here: although the phrase $s \notin f(s)$ has a negation, but not a true self-reference. On the other hand, Russull's famous paradox considers

$$T = \{S \mid S \text{ is a set with } S \notin S\},$$

and this does lead to a paradox in "naive set theory," and had people looking for a type of set theory that avoided this paradox; the usual fix was that formulas such as "the set of all sets such that etc." yields a "class" that may not be a set (intuitively because it may be "too large"). In brief: "$S \notin S$" historically created a paradox and some rethinking of foundations, but "$s \notin f(s)$" gives you a (Cantor's) theorem.

6. Dealing with Paradoxes

There are a number of approaches to dealing with paradoxes. They include:

(1) Ignore the paradox. Carry on regardless.
(2) Admit the paradox, but claim it doesn't matter in practice.
(3) State the paradox in very precise terms and consider the consequences.

For example, when Russell pointed out his paradox (6) of the last section, many mathematicians carried on with whatever they were doing, regardless; however, this paradox did lead some mathematicians to formulate axioms of set theories where

[22]Thanks to Benjamin Israel for pointing out an earlier inaccuracy.

this paradox does not occur. In this course we aim for approach (3), which can lead to a number of results, such as:

(1) The paradox goes away when things are stated precisely.
(2) The paradox doesn't go away, and you have to change your theory if you want to free it of this particular paradox.
(3) The paradox goes away, but only provided that X is true. Then you have proved that X is true (assuming that you don't have paradoxes or related problems in what you are doing).

As examples: the Berry paradox (3) of the last section goes away when things are stated precisely; Russell's paradox (6) lead to a rewriting of set theory with "sets" and "classes" (which includes things "larger than sets"), in which "the set of all sets such that blah blah blah" is a class but not necessarily a set. The halting problem, paradox (7), is an example of a "paradox" that is not really a paradox: it shows you that a certain assumption leads to a "paradox" or "contradiction," and hence the assuption is incorrect; so paradox (7) proves that the "halting problem" cannot be solved by an "algorithm."

7. Countable Sets

It is conceptually helpful to note that for any alphabet Σ, the set Σ^* is *countably infinite*, while the set Power(Σ^*) is *uncountable* (therefore "larger"). This gives another way of understanding that there is no surjection $\Sigma^* \to \text{Power}(\Sigma^*)$.

This is explained in Section 4.2 of [Sip].

7.1. **Countably Infinite Sets.**

Definition 7.1. A set S is *countably infinite* if there is exists bijection $\mathbb{N} \to S$. A set is *countable* if it is finite or countably infinite; a set is *uncountable* if it is not countable.

Example 7.2. For example, the set of integers

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

is countable: indeed, there is a bijection $f\colon \mathbb{N} \to \mathbb{Z}$ with

$$f(1) = 0,\ f(2) = 1,\ f(3) = -1,\ f(4) = 2,\ f(5) = -2,\ f(6) = 3, f(7) = -3, \ldots$$

as indicated above; more precisely, f is given by $f(k) = (k-1)/2$ if k is odd, and $f(k) = k/2$ if k is even. The map $n \mapsto s_n$ gives a bijection Note that $\mathbb{N}$ is a proper subset of $\mathbb{Z}$.

For finite sets $S' \subset S$ with $S' \neq S$, there can never be a bijection from $S' \to S$; hence for infinite sets any such intuition requires some "getting used to." [We will recall a famous quote by John von Neumann regarding this.]

Example 7.3. Let $\Sigma = \{a\}$. Then $\Sigma^* = \{\epsilon, a, a^2, a^3, \ldots\}$, and hence the function $f\colon \mathbb{N} \to \{a\}^*$ taking n to a^{n-1} is a bijection.

Example 7.4. Let $\Sigma = \{a, b\}$. It is not hard to prove that we may list Σ^* as an infinite sequence

$$s_1 = \epsilon, s_2 = a, s_3 = b, s_4 = aa, s_5 = ab, s_6 = ba, s_7 = bb, s_8 = aaa, \ldots$$

in order of increasing length, and secondarily in lexicographical order (such that each string in Σ^* occurs exactly once). Assuming we have proven this, we get a map $n \to s_n$ that is a bijection $\mathbb{N} \to \Sigma^*$. One can similarly prove this for any alphabet Σ.

Example 7.5. Let S be any countably infinite set. Then there is a bijection $S \to \mathbb{N}$ (which is, in particular, a surjection), and hence, by Theorem 3.6 or 3.8 there is no surjection $\mathbb{N} \to \mathrm{Power}(S)$. Hence $\mathrm{Power}(S)$ is uncountable.

Here are some additional examples that we will likely discuss in class.

(1) for any alphabet, Σ, Σ^* is countably infinite (see above), and therefore (see above) $\mathrm{Power}(\Sigma^*)$ is uncountable;
(2) the set, $\mathbb{Q}$, of rational numbers is countably infinite;
(3) if S is countable, for any bijection $S \to T$, T is countable; the same holds with "countable" replaced both times with "uncoutable" (and "finite" and "countably infinite");
(4) Cantor's theorem implies that if S is any infinite set, then $\mathrm{Power}(S)$ is uncountable;
(5) the set $\mathbb{R}$, i.e., of real numbers, is uncountable; this is often proven by "diagonalization," which is essentially the same as (or extremely similar to) the proof of Cantor's theorem; the set of maps $S \to \{0,1\}$ has a simple bijection to the set of all subsets of S, and similarly with $\{0,1\}$ replaced by $\{\texttt{no}, \texttt{yes}\}$ or any two-element set. See [Sip], Chapter 4.

We will also use some facts about bijections, surjections, and injections. Some of these are not intuitive, and some reasonably sounding assertions are false or not necessarily true.

Remark 7.6. If Σ is a fixed alphabet, and $S \subset \Sigma^*$, then S is either finite or countably infinite (we will likely discuss this in class, at least in "naive terms"). For all infinite $S \subset \Sigma^*$, is there necessarily a Python program that can compute a bijection $g \colon \mathbb{N} \to S$? [Exercise.]

Remark 7.7. If $S \subset \mathbb{N}$ and there is no bijection $S \to \mathbb{N}$, then we will show that S is finite. Say that $S \subset \mathrm{Power}(\mathbb{N})$ and that there is no bijection $S \to \mathrm{Power}(\mathbb{N})$, is there necessarily true that S is countable? Is the answer obvious?[23].

8. Undecidability, Acceptance, Halting, and Delightful Programs

In Section 4, we proved that in the context of Python programs,

$$T = \{p \in \Sigma^*_{\mathrm{ASCII}} \mid p \notin \mathrm{LanguageRecBy}(p)\}$$

is unrecognizable, and concluded that (1) NON-ACCEPTANCE is unrecognizable, and therefore (2) ACCEPTANCE is undecidable.

In this section we give the more common argument that shows (1) ACCEPTANCE is undecidable, and therefore (2) NON-ACCEPTANCE is unrecognizable; this is done in Section 4.2 of [Sip].

However, our proof that ACCEPTANCE is undecidably is different — at least in spirit — from the usual proof in that:

[23]The assumption that S is countably under these conditions is called *the continuum hypothesis*; it was a long-standing open problem if the standard set theory axioms (i.e., ZFC, which assumes the Axiom of Choice) imply the continuum hypothesis; in roughly 1963, Paul Cohen settled this negatively, using *forcing arguments* to prove that the continuum hypothesis is *independent* of ZFC.

(1) We define a *delightful program* to be any program that recognizes ACCEPTANCE; such programs exist in many situations and can be built from *universal programs*, e.g., *universal Turing machines* in the context of Turing machines, *universal Python programs* (or certain *Python debuggers*) in the context of Python programs, etc.
(2) For any delightful program, we construct an input on which the delightful program "loops" in the sense that it does not (halt and) answer "yes" or "no."
(3) As an immediate consequence, the ACCEPTANCE problem is undecidable.

Since most of the work in proving (1)–(3) is in part (2), we spend most of our time proving a "true result" about certain programs — *delightful programs* — which do exist, and derive the undecidability of the acceptance problem as an immediate consequence[24]. However, our proof of (2) above is essentially the usual argument, i.e., that in Section 4.2 of [Sip].

It will be convenient for us to prove the above theorem in a very general context, especially when we later discuss oracle machines. We call the general context a *yes/no/loops systems*, which is a generalization of the yes/no tables of Sections 2 and 3.

In the second subsection we do the above restricted to the context of Turing machines; this subsection closely resembles part of Section 4.2 of [Sip].

8.1. **Delightful Programs and Undecidability in Yes/No/Loops Systems.**

Definition 8.1. By a *yes/no/loops system* we mean a triple $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R, \mathrm{EncodeP}, \mathrm{EncodeBoth})$ such that

(1) $\mathcal{P}, \mathcal{I}$ are sets—the *programs* and *inputs*;
(2) $R\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}, \texttt{loops}\}$ and is called the *result function*,
(3) EncodeP is an injection $\mathcal{P} \to \mathcal{I}$ called the *program encoding*,
(4) EncodeBoth is an injection $\mathcal{P} \times \mathcal{I} \to \mathcal{I}$ called the *program and input encoding*.

For brevity we write $\langle p \rangle$ for $\mathrm{EncodeP}(p)$, and $\langle p, i \rangle$ for $\mathrm{EncodeBoth}(p, i)$. [There is no ambiguity since the comma "," distinguishes between $\langle p \rangle$ and $\langle p, i \rangle$.] Similarly, for brevity we use the notation $\mathcal{S} = \big(\mathcal{P}, \mathcal{I}, R, \langle \cdot \rangle, \langle \cdot, \cdot \rangle\big)$ for a yes/no/loops system.

Notice that Section 4.2 of [Sip] uses the same notation $\langle\ \rangle$ and $\langle\ ,\ \rangle$.

Example 8.2. Fix some conventions regarding valid Python programs, such that no Python program contains a symbol $\sigma_0 \in \Sigma_{\mathrm{ASCII}}$. Then we may take $\mathcal{P} = \mathcal{I} = \Sigma^*_{\mathrm{ASCII}}$, where $\langle p \rangle$ is p itself, and set $\langle p, i \rangle = p\sigma_0 i$.

Example 8.3. Say that in the above example all symbols $\sigma_0 \in \Sigma_{\mathrm{ASCII}}$ can occur in a valid Python program. Then we can no longer take $\langle p, i \rangle$ to $p\sigma_0 i$, and the encoding $\langle p, i \rangle$ needs a way to describe when p ends and i begins (i.e., the map $\langle p, i \rangle \overset{\mathrm{def}}{=} p\sigma_0 i$

[24]Here we acknowledge a discussion with Yuval Peres, where Yuval emphasized to us the merit of proving a "true result" and showing a non-existence theorem as a corollary. For example, one can prove that that there are infinitely many primes $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$ by assuming that only finitely many exist, say p_i is the last, and considering $p_1 \ldots p_i + 1$. But it is not much harder to show that $\sum_i 1/p_i = \infty$ (I know of two similar proofs), which is a "true result," and immediately implies that there are infinitely many primes.

may no longer be an injection). In this case, for any string $s = \sigma_1 \ldots \sigma_k$ of length k, define Duplex(s) to be the string of length $2k$ given by

$$\mathrm{Duplex}(s) = \sigma_1\sigma_1\sigma_2\sigma_2\sigma_3 \ldots \sigma_{k-1}\sigma_k\sigma_k.$$

Let $\langle p, i\rangle = \mathrm{Duplex}(p)ab\,i$: we can detect when p ends, and recover p as $p = \sigma_1\sigma_3 \ldots \sigma_{2m-3}$ for the smallest $m \in \mathbb{N}$ such that $\sigma_{2m-1} \neq \sigma_{2m}$.

Example 8.4. We can restrict the discussion of Turing machines to "standardized Turing machines" as discussed in class; in this way, and Turing machine, M, can be expressed a string $\langle M\rangle$, over a fixed alphabet, such as $\{0, 1, \#\}$ (with # a separator and $0, 1$ used to express natural numbers); similarly inputs, i, become a subset of $\{0, 1, \#\}^*$. Hence we set $\mathcal{P} = \mathcal{I} = \{0, 1, \#\}^*$, and if $p \in \mathcal{P}$ represents a valid Turing machine, and $i \in \mathcal{I}$ is a valid input to p it makes sense of whether or not p accepts i (in which case $R(p, i) = \texttt{yes}$), or p rejects i (in which case $R(p, i) = \texttt{no}$), or something else happens to p on input i (in which case $R(p, i) = \texttt{loops}$, although this does not imply that p is necessarily stuck on some infinite loop). If p is not a valid Turning machine description, or i is not a valid input to p, one can adapt the convention that $R(p, i)$ is no, although often this convention does not matter.

Example 8.5. Let Σ be an alphabet, and $A \subset \Sigma^*$. Then one can speak of a "Turing machines with oracle A," that for a fixed A gives a yes/no/loops system. Similarly for "Python program with oracle A," etc.

We now define recognizable languages in the same way as we did for yes/no systems; however, there is a new notion of *decidable languages*.

Definition 8.6. Let $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R, \langle\cdot\rangle, \langle\cdot, \cdot\rangle)$ be a yes/no/loops system. For each $p \in \mathcal{P}$, the *language recognized by* p is defined to be

$$\mathrm{LanguageRecBy}(p) = \{i \in \mathcal{I} \mid R(p, i) = \texttt{yes}\} \subset \mathcal{I};$$

we say a subset $L \subset \mathcal{I}$ is *recognizable (in the systems* $S = (\mathcal{P}, \mathcal{I}, R)$ if $L =$ Recgonizes(p) for some $p \in \mathcal{P}$. We define

$$\mathrm{ACCEPTANCE}_S \overset{\mathrm{def}}{=} \{\langle p, i\rangle \mid R(p, i) = \texttt{yes}\},$$

and

$$\mathrm{HALT}_S \overset{\mathrm{def}}{=} \{\langle p, i\rangle \mid R(p, i) \in \{\texttt{yes}, \texttt{no}\}\}.$$

By a *decider* we mean a $p \in \mathcal{P}$ such that $R(p, i) \in \{\texttt{yes}, \texttt{no}\}$ for all $i \in \mathcal{I}$; we say that $L \subset \mathcal{I}$ is *decidable* if some decider recognizes L.

We also define the *negation* function, denoted $\neg$, as

$$\neg\texttt{no} = \texttt{yes},\ \neg\texttt{yes} = \texttt{no},\ \neg\texttt{loops} = \texttt{loops};$$

we easily see that $\neg\neg v = v$ for all $v \in \{\texttt{yes}, \texttt{no}, \texttt{loops}\}$.

Definition 8.7. Let $\mathcal{S} = \bigl(\mathcal{P}, \mathcal{I}, R, \langle\cdot\rangle, \langle\cdot, \cdot\rangle\bigr)$ be a yes/no/loops system. If $p \in \mathcal{P}$, we call a $q \in \mathcal{P}$ a *mysterious counterpart of* p if

$$\forall m \in \mathcal{P}, \quad R\bigl(q, \langle m\rangle\bigr) = \neg R\Bigl(p, \bigl\langle m, \langle m\rangle\bigr\rangle\Bigr).$$

We say that a $p \in \mathcal{P}$ is *delightful* if is recognizes $\mathrm{ACCEPTANCE}_{\mathcal{S}}$.

For example, a universal Python program is delightful in this context, as is a universal Turing machine in the context of Turing machines. Of course, a delightful program can try to determine if p halts on input i by a number of methods, and if none of these work (and they all eventually terminate) then afterwards one can run a universal machine.

Algorithmically, it is straightforward to take any Turing machine (any Python program, etc.), p, and construct a mysterious version of p, by (1) checking if the input is of the form $\langle m\rangle$ for some $m \in \mathcal{P}$, then (2) running p on input $\langle m, \langle m\rangle\rangle$, then (3) negating the answer. This is true in the above examples, and true in similar examples when $\langle\rangle$ and $\langle,\rangle$ and their inverses can be computed by some algorithm.

Hence the term *mysterious* does not refer to the difficulty in its construction, but rather in the somewhat mysterious theorem it proves.

Theorem 8.8. *Let $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R, \langle\cdot\rangle, \langle\cdot,\cdot\rangle)$ be a yes/no/loops system. Say that $h \in \mathcal{P}$ is a delightful program that has a mysterious counterpart $d \in \mathcal{P}$. Then:*

(1) $R(d, \langle d\rangle) = \texttt{loops}$; and
(2) $R(h, \langle d, \langle d\rangle\rangle) = \texttt{loops}$.

In particular, h is not a decider.

Proof. If (1) is not true, then $R(d, \langle d\rangle)$ is either `yes` or `no`; we will derive a contradiction in either case (very similar to Cantor's theorem): since

$$\forall m \in \mathcal{P}, \quad R(d, \langle m\rangle) = \neg R(h, \langle m, \langle m\rangle\rangle),$$

we have

$$R(d, \langle d\rangle) = \neg R(h, \langle d, \langle d\rangle\rangle). \tag{22}$$

Assume that $R(d, \langle d\rangle) = \texttt{yes}$: then $R(h, \langle d, \langle d\rangle\rangle) = \neg\texttt{yes} = \texttt{no}$, and since h recognizes ACCEPTANCE, $R(d, \langle d\rangle)$ cannot equal `yes`. But this contradicts the assumption that $R(d, \langle d\rangle) = \texttt{yes}$. We argue similarlly if we assume $R(d, \langle d\rangle) = \texttt{no}$. Hence $R(d, \langle d\rangle) = \texttt{loops}$.

(2) follows from (1) and (22). □

Corollary 8.9. *Let $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R, \langle\cdot\rangle, \langle\cdot,\cdot\rangle)$ be a yes/no/loops system such that each program has a mysterious counterpart. Then any delightful program, d, must loop on input $\langle h, \langle h\rangle\rangle$ where h is a mysterious version of d. In particular,* ACCEPTANCE$_{\mathcal{S}}$ *is undecidable.*

8.2. **Delightful Turing Machines and Undecidability.**

Definition 8.10. We say that a Turing machine is *delightful* if it recognizes the language

$$\mathrm{A_{TM}} = \mathrm{ACCEPTANCE_{TM}} = \{\langle M, i\rangle \mid M \text{ accepts } i\}.$$

For example, a universal Turing machine is delightful. As another example, given the input $\langle M, i\rangle$, you could run certain subroutines to determine if M accepts i, and if these subroutines do not succeed, then run a universal Turing machine: for example, your subroutine might check whether or not $\langle M\rangle$ is a valid Turing machine, and, if so, whether or not its δ-function ever transitions to the state q_{accept}. There are, of course, more sophisticated tests to try—a lot of practical algorithms (excluding some video games and electronic media) have a structure that makes it easy to verify that they always halt.

There might also be some algorithm that is delightful, i.e., that recognizes $\mathrm{ACCEPTANCE}_{\mathrm{TM}}$, for reasons that we do not understand (or are, moreover, unprovable).

By the definition of how a Turing machine works, on any input, a Turing machine computation results in either: (1) halting in q_{accept}, (2) halting in q_{reject}, or (3) never halting. We define the *opposite result* of (1) to be (2), and of (2) to be (1), and of (3) to be (3) (hence the opposite result of never halting is, again, never halting).

Definition 8.11. If H is any Turing machine, we say that D is a *mysterious form of* H if for all inputs to D of the form $\langle M\rangle$, the result of D is the opposite result of H on input $\langle M, \langle M\rangle\rangle$. [Hence we don't require anything about how D behaves on inputs that are not of the form $\langle M\rangle$.]

You should convince yourself that there is a mysterious form of any Turing machine. The term *mysterious* refers to the theorem below.

Theorem 8.12. *Let H be a delightful Turing machine, and D a mysterious form of H. Then:*

(1) D on input $\langle D\rangle$ must loop (i.e., never terminates in either q_{accept} or q_{reject}); and
(2) H on input $\langle D, \langle D\rangle\rangle$ must loop.

Proof. Assume that D on input $\langle D\rangle$ terminates in q_{accept}; let us derive a contradiciton: since D is a mysterious form of H, H rejects $\langle D, \langle D\rangle\rangle$. But since H recognizes the acceptance problem, this implies that D does not accept $\langle D\rangle$, which contradicts our assumption.

Similarly the assumption that D terminates in q_{reject} results in a contradiction.

Hence D loops on input $\langle D\rangle$, proving statement (1) of the theorem. Statement (2) follows immediately from (1) and the fact that D is a mysterious version of H. □

Notice that the above theorem is almost identical to the standard proof that the acceptance problem is undecidable (see also Section 4.2 of [Sip]); however, this theorem proves a result about Turing machines, H, that actually exist, rather than merely proving that a certain type of Turing machine does not exist.

Corollary 8.13. *The acceptance problem is undecidable (in the context of Turing machines).*

Proof. If the acceptance problem were decided by H, then H would not loop on any input, contradicting Theorem 8.12. □

9. EXERCISES

The first subsection of exercises are sample problems with solutions, to indicate the level of detail expected in homework solutions.

Subsections 9.5 and 9.6 will not be covered until we discuss Turing machines in Chapter 3 of [Sip].

9.1. **Sample Exercises with Solutions.**

People often ask me how much detail they need in giving explanations for the homework exercises. Here are some examples. The material in brackets [like this] is optional.

Sample Question Needing a Proof: If $f\colon S \to T$ and $g\colon T \to U$ are surjective (i.e., onto) is $g \circ f$ (a map $S \to U$) is necessarily surjective? Justify your answer.

Answer: Yes.

[To show that $g \circ f$ is surjective, we must show that if $u \in U$, then there is an $s \in S$ such that $(g \circ f)(s) = u$.]

If $u \in U$, then since g is surjective there is a $t \in T$ such that $g(t) = u$. Since f is surjective, there is an $s \in S$ such that $f(s) = t$. Hence

$$(g \circ f)(s) = g(f(s)) = g(t) = u.$$

Therefore each $u \in U$ is $g \circ f$ applied to some element of S, and so $g \circ f$ is surjective.

Sample Question Needing a Counterexample: If $f\colon S \to T$ is injective, and $g\colon T \to U$ is surjective, is $g \circ f$ is necessarily injective? Justify your answer.

Answer: No.

[To show that $g \circ f$ is not necessarily injective, we must find one example of such an f and g where $g \circ f$ is not injective.]

Let $S = T = \{a, b\}$ and $U = \{c\}$; let $f\colon S \to T$ be the identity map (i.e., $f(a) = a$ and $f(b) = b$), and let $g\colon T \to U$ (there is only one possible g in this case) be given by $g(a) = g(b) = c$.

Then f is injective (since $f(a) \neq f(b)$) and g is surjective, since $U = \{c\}$ and $c = g(a)$). However $g \circ f$ is not injective, since $(g \circ f)(a) = c = (g \circ f)(b)$.

Injectivitiy and Surjectivity of a Given Map: If $f\colon \mathbb{N} \to \mathbb{N}$ is given by $f(n) = 2n + 5$, is f injective? Is f surjective?

Answer: f is injective, because if $f(n_1) = f(n_2)$, then $2n_1 + 5 = 2n_2 + 5$ and therefore $n_1 = n_2$.

[Hence f maps distinct values of $\mathbb{N}$ to distinct values of $\mathbb{N}$, i.e., $n_1 \neq n_2$ implies that $f(n_1) \neq f(n_2)$.]

f is not surjective, because there is no value $n \in \mathbb{N}$ such that $f(n) = 1$: if such an n existed, then $2n + 5 = 1$ and so $n = -2$ which is not an element of $\mathbb{N}$.

Level of detail for Cantor's Theorem Exercise: Let $S = \{1, 2, 3, 4\}$ and $f\colon S \to \mathrm{Power}(S)$ be given by $f(1) = \{1, 2\}$.

(1) Given this information, what can you say about What is $T = \{s \mid s \notin f(s)\}$?
(2) Give a direct argument (without using Cantor's theorem) that $f(1) \neq T$.

Answer:

(1) Since $1 \in f(1)$, it is not true that $s \notin f(s)$ for $s = 1$, and hence $1 \notin T$. (You could write this slightly shorter: since $1 \in f(1)$, it is not true that $1 \notin f(1)$, and hence $1 \notin T$.)
(2) Since $1 \notin T$ but $1 \in f(1)$, we have $T \neq f(1)$.

9.2. Exercises for Sections 2–3: Cantor's Theorem, its Generalizations, and Recognizable/Decidable Languages.

Exercise 9.2.1. Let $S = \{1, 2, 3\}$ and $f : S \to \mathrm{Power}(S)$ be given by

$$f(1) = \{1, 2\},\ f(2) = \{1, 3\},\ f(3) = \{2, 3\}.$$

What is $T = \{s \mid s \notin f(s)\}$?

Exercise 9.2.2. Let $S = \{1, 2, 3\}$ and $f : S \to \mathrm{Power}(S)$ satisfy $f(1) = \{1, 2\}$. Let $T = \{s \mid s \notin f(s)\}$.

9.2.2(a) Without using Cantor's theorem, give a direct argument to show that $T \neq \{1, 2\}$ (knowing only that $f(1) = \{1, 2\}$).

9.2.2(b) Without any additional information, can you determine whether or not $2 \in T$? To answer this question, you should either (1) prove that $2 \in T$, (2) prove that $2 \notin T$, or (3) give an example of an f such that $2 \in T$ and another example where $2 \notin T$.

Exercise 9.2.3. Let $S = \{a, b, c\}$ and let $f : S \to \mathrm{Power}(S)$ any function such that

$$a \notin f(a), \quad b \notin f(b), \quad c \notin f(c).$$

9.2.3(a) Explain why $f(a)$ cannot be all of S (argue directly, without appealing to Cantor's theorem).

9.2.3(b) Similarly, explain why none of $f(b), f(c)$ equal S.

9.2.3(c) What is the set

$$T = \{s \in S \mid s \notin f(s)\}?$$

Exercise 9.2.4. Let $S = \{a, b, c\}$ and let $f : S \to \mathrm{Power}(S)$ any function such that

$$a \notin f(a), \quad b \in f(b), \quad c \notin f(c).$$

9.2.4(a) Explain why $f(b)$ cannot equal $\{a, c\}$ (argue directly, without appealing to Cantor's theorem).

9.2.4(b) Similarly, explain why none of $f(a), f(c)$ can equal $\{a, c\}$.

9.2.4(c) What is the set

$$T = \{s \in S \mid s \notin f(s)\}?$$

Exercise 9.2.5. Let $S = \{1, 2, 3\}$ and $f : S \to \mathrm{Power}(S)$ be given some map. Can it be that

$$T = \{s \mid s \in f(s)\}$$

lies in the image of f? [Either give an example of such an f, or explain why it doesn't exist. You should give an explanation "from scratch," without relying on Cantor's theorem or any other result from class or these notes.]

Exercise 9.2.6. Let $f : A \to B$ and $g : B \to C$ be functions on sets A, B, C. Consider the composition $gf : A \to C$ of f and g (you may have seen gf written as $g \circ f$ for clarity).

9.2.6(a) Say that gf is injective. Is f necessarily injective? Explain (give a proof that f is injective or give examples of f, g where f is not injective).

9.2.6(b) Say that gf is injective. Is g necessarily injective?

9.2.6(c) Say that gf is injective and, in addition, $|B| \leq |A|$ and A, B are finite sets. Is g necessarily injective?

9.2.6(d) Say that gf is surjective. Is f necessarily surjective?
9.2.6(e) Say that gf is surjective. Is g necessarily surjective?

Exercise 9.2.7. Let $f\colon \mathbb{N} \to \mathrm{Power}(\mathbb{N})$ be given by

$$f(n) = \{m \in \mathbb{N} \mid m + n/2 \text{ is a perfect square}\} = \{m \in \mathbb{N} \mid m + n/2 = k^2 \text{ for some } k \in \mathbb{N}\}$$

What is $T = \{s \mid s \notin f(s)\}$?

Exercise 9.2.8. A department has 3 profs, $P = \{A, B, C\}$. It is given that

Prof. A: thinks that no one in the department works too much, and
Prof. C: thinks that everyone in the department works too much.

For $x \in P$, let

$$f(x) = \{y \in P \mid x \text{ thinks that } y \text{ works too much.}\}$$

9.2.8(a) What does the above information tell you about f ?
9.2.8(b) What can you say about

$$T = \{s \in P \mid s \text{ thinks that } s \text{ does not work too much}\},$$

and how do you know that $T \neq f(A)$ and $T \neq f(C)$?
9.2.8(c) Now say that, in addition, you know that
Prof. B: thinks that Profs. A and C work too much, but not themself.
What is

$$T = \{s \in P \mid s \text{ thinks that } s \text{ does not work too much}\}?$$

Exercise 9.2.9. A department has 3 profs, $P = \{A, B, C\}$. It is given that

Prof. A: thinks that Prof. B works too much,
Prof. B: thinks that Prof. C works too much, and
Prof. C: thinks that Prof. A does not work too much.

Find a bijection $g\colon P \to P$ such that

$$T = \{s \in P \mid s \text{ thinks that } g(s) \text{ does not work too much}\}$$

can be determined. Then state this as an instance of generalized Cantor's theorem.

Exercise 9.2.10. A department has 3 profs, $P = \{A, B, C\}$. It is given that

Prof. A: thinks that no one in the department works too much,
Prof. B: thinks that Profs. A and C work too much, but not themself, and
Prof. C: thinks that everyone in the department works too much.

Describe

$$T = \{s \mid s \text{ thinks that } s \text{ does not work too much}\}.$$

Is there a prof who thinks that the elements of T work too much, but not the elements of $P \setminus T$?

Exercise 9.2.11. A department has 3 profs, $P = \{A, B, C\}$, who each have access to three foods, $F = \{\text{hummus}, \text{falafel}, \text{pita}\}$. It is given that

Prof. A and B: like and dislike the same foods, and
Prof. C: likes hummus and falafel, but dislikes pita.

Can you descibe a subset

$$T \subset \{\text{hummus}, \text{falafel}, \text{pita}\}$$

such that no prof likes the foods in T, and dislikes the other foods (i.e., those in $F \setminus T$)? Exaplain.

Exercise 9.2.12. Complete the proof of Theorem 3.8.

Exercise 9.2.13. The Rose family has four people: Johnny, Moira, David, and Alexis. Let R be the set consisting of these four people, i.e.,

$$R = \{\text{Johnny}, \ \text{Moira}, \ \text{David}, \ \text{Alexis}\}.$$

It is given that:

Johnny: loves everyone;
Moira: loves (and only loves) Jonny and Moira;
David: loves no one; and
Alexis: loves (and only loves) David and Alexis.

Let

$$T = \{r \in R \mid r \text{ does not love themself}\} = \{r \in R \mid r \text{ does not love } r\},$$

i.e., T is the subset of R that consists of each person who does not love themself[25].

9.2.13(a) What is T? In other words, list the elements between braces ($\{,\}$).
9.2.13(b) Explain why if David does not love themself, then the set T cannot equal the set of people whom David loves, i.e., the empty set, regardless of whom anyone else loves.

Exercise 9.2.14. Same as Exercise 9.2.13, with the modification that

Johnny: loves (and only loves) Johnny and Moira;
Moira: loves everyone;
David: loves no one; and
Alexis: loves no one.

Exercise 9.2.15. Same as Exercise 9.2.13, with the modification that no one loves anyone.

Exercise 9.2.16. Consider the setting in Exercise 9.2.13, with the modification that everyone loves everyone.

9.2.16(a) What is T?
9.2.16(b) Explain why if David loves themself, then the set T cannot equal the set of people whom David loves, i.e., all of R, regardless of whom anyone else loves.

Exercise 9.2.17. A village has five residents: Martin, Short, Gomez, Colbert, and Batiste. Let V be the set consisting of these five people, i.e.,

$$V = \{\text{Martin}, \ \text{Short}, \ \text{Gomez}, \ \text{Colbert}, \ \text{Batiste}\}.$$

It is given that:

Martin: thinks that Martin and Short are old, and the rest are not old;
Short: thinks that Martin is old, and the rest are not old;

[25]We thank Sophie MacDonald who pointed out to us this singular, gender neutral form in Fall 2021.

Gomez: thinks that Martin, Short, and Colbert are old, and the rest are not old;
Colbert: thinks that Martin and Short are old, and the rest are not old; and
Batiste: thinks that no one is old.

$$S = \{v \in V \mid v \text{ does not think of themself as old}\},$$

9.2.17(a) What is S?
9.2.17(b) Explain why if Martin thinks of themself as old, then S does not equal the subset of V whom Martin thinks are old, regardless of what anyone else thinks.
9.2.17(c) Explain why if Batiste thinks that no one is old, then S does not equal the subset of V whom Batiste thinks are old, regardless of what anyone else thinks.

Exercise 9.2.18. Consider the same situation as Exercise 9.2.17. Let $f\colon V \to V$ be the function (map, morphism, etc.) given by:

$$f(\text{Martin}) = \text{Short}, \quad f(\text{Short}) = \text{Gomez}, \quad f(\text{Gomez}) = \text{Colbert},$$

$$f(\text{Colbert}) = \text{Batiste}, \quad f(\text{Batiste}) = \text{Martin}.$$

(Notice that f is a bijection, and therefore has an inverse function, f^{-1}.) Let

$$S = \{v \in V \mid v \text{ does not think of themself as old}\},$$

and

$$S' = \{v \in V \mid v \text{ does not think of } f(v) \text{ as old}\}.$$

9.2.18(a) Explain why if Gomez does not think that Gomez, themself, is old, then the set S above does not equal the set of people whom Gomez thinks are old, regardless of what anyone else thinks.
9.2.18(b) Explain why if Gomez thinks that Colbert is old, then the set S' above does not equal the set

$$S'' = \{v \in V \mid v \text{ thinks of } f(v) \text{ as old}\},$$

regardless of what anyone else thinks.
9.2.18(c) Explain why if Batiste thinks that no one is old, then both sets S and S' above do not equal the set of people whom Batiste thinks are old, regardless of what anyone else thinks.
9.2.18(d) If $f\colon V \to V$ were any other function—not necessarily a bijection—would part (c) still be true?

Exercise 9.2.19. Let $L \subset \Sigma^*_{\text{ASCII}}$ be decidable by a Python program. Is L necessarily recognizable? Is $\Sigma^*_{\text{ASCII}} \setminus L$ necessarily recognizable?

Exercise 9.2.20. Let $L \subset \Sigma^*_{\text{ASCII}}$ be recognizable by a Python program. Is L necessarily decidable? Is $\Sigma^*_{\text{ASCII}} \setminus L$ necessarily recognizable?

Exercise 9.2.21. Let

$$L = \{p \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that halts on at least three distinct inputs to } p\}.$$

Is p decidable? Is p recognizable?

Exercise 9.2.22. Which of the following maps are injections (i.e., one-to-one), and which are surjections (i.e., onto)? Justify your answer (briefly); the justification should look like the answers to similar questions in Subsection 9.1.

9.2.22(a) $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(x) = x + 1$.
9.2.22(b) $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(x) = x^2$.
9.2.22(c) $f\colon \mathbb{Z} \to \mathbb{Z}$ given by $f(x) = x + 1$.
9.2.22(d) $f\colon \mathbb{Z} \to \mathbb{Z}$ given by $f(x) = x^2$.

Exercise 9.2.23. If $f\colon S \to T$ and $g\colon T \to U$ are both injective (i.e., one-to-one), is $g \circ f$ (which is a map $S \to U$) necessarily injective? Justify your answer (briefly); the justification should look like the answers to similar questions in Subsection 9.1.

Exercise 9.2.24. Let

$$S = \{\text{Oppenheimer}, \text{Barbie}, 2001, \text{Encounters}\}, \qquad S' = \{A, B, C, D\}.$$

Say that:

(1) Student A has seen the movie "Oppenheimer;"
(2) Student B has not seen the movie "Barbie;"
(3) Student C has not seen the movie "Encounters at the End of the World;"
(4) Student D has seen the movie "2001: A Space Odyssey;" and
(5) You don't have any additional information.

For each $x \in S'$, let $f(x)$ be the movies that Student x has seen; hence f is a function $f\colon S' \to \text{Power}(S)$.

9.2.24(a) What can you assert about $f(A)$? What do you NOT know about $f(A)$?
9.2.24(b) Give a surjection $g\colon S \to S'$ such that for each $x \in S$ you can answer the question "is x in $f(g(x))$?"
9.2.24(c) Say that x is a student, perhaps one of A, B, C, D, but perhaps a different student. Construct a subset $T \subset S$ such that if x seen the movies in T and has not seen the movies not in T, then x cannot equal any of A, B, C, D.

Exercise 9.2.25. Let

$$S = \{\text{Oppenheimer}, \text{Barbie}, 2001, \text{Encounters}\}, \qquad S' = \{A, B, C, D\}.$$

Say that you know that:

(1) the movie "Oppenheimer" was seen by A, B, C and not by D;
(2) the movie "Barbie" was seen by A, C and not by B, D;
(3) the movie "2001: A Space Odyssey" was seen by C;
(4) the movie "Encounters at the End of the World" was not seen by C.

For each $x \in S'$, let $f(x)$ be the movies that Student x has seen; hence f is a function $f\colon S' \to \text{Power}(S)$. With only the above information, is there a surjection $g\colon S \to S'$ such that for each $x \in S$ you can answer the question "is x in $f(g(x))$?" Explain. [Hint: It may help to draw a graph/diagram with the elements of S on the left, elements of S' on the right, and an arrow from $s \in S$ to $s' \in S'$ if you can answer the question "was s seen by s'?" However, since each of S, S' has only 4 elements, you can probably solve this without a digram, say by considering A, B, D.]

Exercise 9.2.26. Are the following statements true or false? If they are true, explain why; if false, give a counterexample. In these statements, $\Sigma = \Sigma_{\text{ASCII}}$, and $L_1, L_2 \subset \Sigma^* = \Sigma^*_{\text{ASCII}}$ are subsets.

9.2.26(a) If L_1 is recognizable, then L_1 is decidable.
9.2.26(b) If L_1 is unrecognizable, then L_1 is undecidable.
9.2.26(c) If L_1 is decidable, then L_1 is recognizable.
9.2.26(d) If L_1 is undecidable, then L_1 is unrecognizable.
9.2.26(e) If L_1, L_2 are decidable, then $L_1 \cup L_2$ is decidable.
9.2.26(f) If L_1, L_2 are undecidable, then $L_1 \cup L_2$ is undecidable.
9.2.26(g) If L_1, L_2 are recognizable, then $L_1 \cup L_2$ is recognizable.
9.2.26(h) If L_1, L_2 are unrecognizable, then $L_1 \cup L_2$ is unrecognizable.
9.2.26(i) If L_1 is decidable, then $\Sigma^* \setminus L_1$ is decidable.
9.2.26(j) If L_1 is recognizable, then $\Sigma^* \setminus L_1$ is recognizable.
9.2.26(k) If L_1 is recognizable, then L_1 is decidable.
9.2.26(l) If L_1, L_2 are decidable, then $L_1 \setminus L_2$ is decidable.
9.2.26(m) If L_1, L_2 are recognizable, then $L_1 \setminus L_2$ is recognizable.

Exercise 9.2.27. For each of the following languages, L, say whether or not L is decidable and whether or not it is recognizable. Here σ_0 is some element of $\Sigma_{\rm ASCII}$ such that no valid Python program contains σ_0 (in class in 2024 we imagined this to be σ_0 equal to $\langle \mathrm{FS} \rangle$, the "file separator" in $\Sigma_{\rm ASCII}$). **Justify your answer (no points are given for an answer without explanation).**

9.2.27(a) The language of strings $p\sigma_0 i$ such that p accepts i after running for 10 steps.
9.2.27(b) The language of strings $p\sigma_0 i$ such that p rejects i.
9.2.27(c) The language of strings $p\sigma_0 i$ such that p halts on input i.
9.2.27(d) The language of strings $p\sigma_0 i$ such that p accepts or loops on input i.
9.2.27(e) The language of valid Python programs, p, such that p rejects at least one input, i.e., at least one $i \in \Sigma^*_{\rm ASCII}$.
9.2.27(f) The language of valid Python programs, p, such that p accepts at least two values of $i \in \Sigma^*_{\rm ASCII}$.
9.2.27(g) The language of valid Python programs, p, such that p accepts all its inputs.

Exercise 9.2.28. [26] Let $i_1, i_2, \ldots$ be a sequence elements of $\Sigma^*_{\rm ASCII}$ such that each element of $\Sigma^*_{\rm ASCII}$ appears exactly once in this sequence.[27] Say that p is a Python program, and we want to know if p accepts at least one input. We can do this by the following algorithm:

Phase 1: simulate p for one step on input i_1;
Phase 2: simulate p for two steps on i_1 and one step on i_2;
Phase 3: simulate p for three steps on i_1, for two steps on i_2, and for one step on i_3;
etc.:
Phase k: on the k-th phase, for $j = 1, 2, \ldots, k$ we simulate p for $k - j + 1$ steps on i_j;

Consider the total number of steps run in each phase; for example, Phase 3 has 6 steps total, and the total number of steps in Phases 1 to 3 is $1 + 3 + 6 = 10$. (Our convention is that when you simulate p on an input for some number of steps,

[26]This question arose in class, September 2023; we thank, in particular, Vishnu Yadavalli for the question, and Ellen Lloyd for an algorithm given below.

[27]In CPSC 421/501, we typically do this by listing the strings according to their length (and lexicographical order for strings of equal length), so that $i_1 = \epsilon$ (which is the single string of length 0), $i_2, \ldots, i_{129}$ are the elements of $\Sigma_{\rm ASCII}$, $i_{130}, \ldots, i_{1+128+128^2}$ are the elements of $\Sigma^2_{\rm ASCII}$, etc.

you forget all previous simulations of p on any input.) Say that p accepts only one input, namely i_ℓ, and that p requires m program steps to do so.

9.2.28(a) Show that the total number of steps until the above algorithm stops (i.e., when it detects that p accepts i_ℓ after m steps) is **exactly**

$$(1/6)(\ell+m)^3+O(1)(\ell+m)^2,$$

where the $O(1)$ refers to an "order 1 term," i.e., a function of ℓ,m that is bounded by a constant for $\ell+m$ sufficiently large. By **exactly** we mean that $(1/6)(\ell+m)^3+O(1)(\ell+m)^2$ is both a lower bound and an upper bound (for different values of $O(1)$).

9.2.28(b) Say that we use the following variant: for all $k\in\mathbb{N}$, the k-the phase consists of simulating k steps of p on each of $i_1,\ldots,i_k$. Show that the total number of steps needed is **exactly**

$$(1/3)(\max(\ell,m))^3+O(1)(\max(\ell,m))^2.$$

9.2.28(c) Say that we use the following variant: for all $k\in\mathbb{N}$, the k-the phase consists of simulating $5k$ steps of p on each of $i_1,\ldots,i_{5k}$. Show that the total number of steps needed is **exactly** $c(\max(\ell,m))^3+O(1)(\max(\ell,m))^2$ for some constant, c. What is c ?

9.2.28(d) Using the previous part, for any constant $c>0$, give a variant of the above algorithm that takes **at most** $c(\max(\ell,m))^3+O(1)(\max(\ell,m))^2$ steps.

Exercise 9.2.29. Continuing with the setup and notation as in the previous problem:

9.2.29(a) Describe a variant of the above algorithm that uses no more than $O(1)(\max(\ell,m))^2$ steps.

9.2.29(b) Prove that there is a constant $c>0$ such that any such algorithm requires at least $c(\max(\ell,m))^2$ steps for $\max(\ell,m)$ sufficiently large, and give such a constant, c. [This implies that there is a $c>0$ for which this holds for all $\ell,m\in\mathbb{N}$, but it is simpler to find a c that holds when $\max(\ell,m)$ is sufficiently large.]

Exercise 9.2.30. A department has 3 profs, $P=\{A,B,C\}$, who each have access to three foods, $Q=\{\text{hummus},\text{falafel},\text{pita}\}$. It is given that

Prof. A: likes pita, and dislikes falafel (and we don't know about hummus),
Prof. B: likes falafel (and we don't know about pita and hummus), and
Prof. C: likes hummus and falafel, but dislikes pita.

9.2.30(a) Write a yes/no table for the question "does Prof. p like food q" (ranging over all $p\in P$ and $q\in Q$)?

9.2.30(b) Describe a surjection $g\colon Q\to P$ such that for all $q\in Q$ one can answer the question "does Prof. $g(q)$ like q?"

9.2.30(c) Use g and Theorem 3.8 to give a

$$T\subset\{\text{hummus},\text{falafel},\text{pita}\}$$

such that no prof likes the foods in T, and dislikes the other foods (i.e., those in $Q\setminus T$).

Exercise 9.2.31. Consider the setting of Exercise 9.2.13. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 9.2.32. Consider the setting of Exercise 9.2.14. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 9.2.33. Consider the setting of Exercise 9.2.15. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 9.2.34. Consider the setting of Exercise 9.2.16. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 9.2.35. One sometimes defines a *universal program* to be a program that recognizes ACCEPTANCE in a given context (e.g., Python programs, Duck programs, Turing machines, etc.) (this is a narrower sense of a *universal Python program* that we described in class). Since no valid Duck program can contain the letter Q, let us set $\sigma_0 = Q$ and define

$$\mathrm{ACCEPTANCE}_{\mathrm{Duck}} = \{p\sigma_0 i \mid p \text{ is a valid Duck program that accepts } i\}.$$

9.2.35(a) Give strings p_1, p_2, i_1, i_2 such that $p_1\sigma_0 i_1$ and $p_2\sigma_0 i_2$ have the same length, but

$$p_1\sigma_0 i_1 \in \mathrm{ACCEPTANCE}_{\mathrm{Duck}}$$

and

$$p_2\sigma_0 i_2 \notin \mathrm{ACCEPTANCE}_{\mathrm{Duck}}.$$

9.2.35(b) Use part (a) to show that there is no Duck program p such that $\mathrm{LanguageRecBy}_{\mathrm{Duck}}(p) = \mathrm{ACCEPTANCE}_{\mathrm{Duck}}$.

Exercise 9.2.36. For each $i \in \Sigma^*_{\mathrm{ASCII}}$ and $p \in \Sigma^*_{\mathrm{ASCII}}$, we defined what it means for p to be a valid Duck program, and whether or not "p accepts i" (in the context of Duck programs). Now say that "p rejects i" if p does not accept i. Hence for each $i \in \Sigma^*_{\mathrm{ASCII}}$ and $p \in \Sigma^*_{\mathrm{ASCII}}$, p either accepts i or rejects i (we do not allow p to "loop" on i). Are the following statements true or false? If they are true, explain why; if false, give a counterexample. In these statements, $\Sigma = \Sigma_{\mathrm{ASCII}}$, $\sigma_0 = \langle \mathrm{FS} \rangle$, and $L_1, L_2 \subset \Sigma^* = \Sigma^*_{\mathrm{ASCII}}$ are subsets.

9.2.36(a) If L_1 is Duck-recognizable, then L_1 is Duck-decidable.
9.2.36(b) If L_1 is Duck-decidable, then $L_1^{\mathrm{comp}} = \Sigma^*_{\mathrm{ASCII}} \setminus L_1$ is Duck-decidable.
9.2.36(c) If L_1, L_2 are Duck-decidable, then $L_1 \cup L_2$ is Duck-decidable.
9.2.36(d) If L_1, L_2 are Duck-decidable, then $L_1 \setminus L_2$ is Duck-decidable.
9.2.36(e) If L_1, L_2 are Duck-undecidable, then $L_1 \cup L_2$ is Duck-undecidable.
9.2.36(f) If L_1, L_2 are Duck-undecidable, then $L_1 \setminus L_2$ is Duck-undecidable.
9.2.36(g) $\mathrm{REJECTION}_{\mathrm{Duck}}$ is Duck-decidable, where

$$\mathrm{REJECTION}_{\mathrm{Duck}} = \{p\sigma_0 i \mid p \text{ is a valid Duck program that rejects } i\}$$

(see the previous exercise for a similar question).
9.2.36(h) $\mathrm{REJECTION}_{\mathrm{Duck}}$ (defined in part (g)) is Python-decidable.
9.2.36(i) $\mathrm{REJECTION}_{\mathrm{Duck}} \cup \mathrm{ACCEPTANCE}_{\mathrm{Duck}}$ is Duck-decidable.

9.2.36(j) If L_1 is Duck-decidable, then L_1 is Python-decidable.
9.2.36(k) If L_1 is Duck-decidable, then $L_1^{\rm comp} = \Sigma^*_{\rm ASCII} \setminus L_1$ is Python-decidable.

Exercise 9.2.37. We often use the following fact to produce undecidable languages: "if L is unrecognizable, then L and $L^{\rm comp} = \Sigma^* \setminus L$ are undecidable." Is the converse true?[28]

Exercise 9.2.38. Show that

$$L = \{q \mid q \text{ is a valid Python program that accepts none of its inputs}\}$$

is unrecognizable in the following way: we know that NON-ACCEPTANCE is unrecognizable; hence it suffices to show we can use a recognizer for L to build a recognizer for NON-ACCEPTANCE. Given a string $p\sigma_0 j$ where p is a valid Python program, build a program q such that $q \in L$ iff $p\sigma_0 j \in$ NON-ACCEPTANCE. What is your procedure for building q from $p\sigma_0 j$? [Hint: You could do something similar to Example 4.32.]

Exercise 9.2.39. Show that

$$L = \{q \mid q \text{ is a valid Python program that accepts all of its inputs}\}$$

is unrecognizable. [Hint: Reduce NON-ACCEPTANCE to L: given $p\sigma_0 i$, let q be the program that on input j simulates p on input i for $|j|$ steps (using a universal Python program) and takes an appropriate action based on the result.]

Exercise 9.2.40. Show that

$$L = \{q \mid q \text{ is a valid Python program that accepts exactly three of its inputs}\}$$

is unrecognizable. [Hint: you can modify slightly the construction of q from $p\sigma_0 j$ in Exercise 9.2.38.]

Exercise 9.2.41. Show that

$$L = \{p\sigma_0 q \mid p \text{ and } q \text{ are valid Python programs that recognize different languages}\},$$

is unrecognizable; you may use the result in any of Example 4.32, Exericse 9.2.38, or Exericse 9.2.39. [Hint: you can take p be a very simple program.]

Exercise 9.2.42. Show that

$$L = \{p\sigma_0 q \mid p \text{ and } q \text{ are valid Python programs that recognize the same language}\}$$

is unrecognizable; you may use the result in any of Example 4.32, Exericse 9.2.38, or Exericse 9.2.39. [Hint: you can take p be a very simple program.]

Exercise 9.2.43. Show that

$$L = \{p\sigma_0 q \mid p \text{ and } q \text{ are valid Python programs that both accept some } i \in \Sigma^*_{\rm ASCII}\}$$

is undecidable but recognizable; you may use the result in any of Example 4.32, Exericse 9.2.38, or Exericse 9.2.39. [Hint: you can take p be a very simple program.]

[28]This question was posed in class, 2024, by Amin Fahiminia.

9.3. **Paradox Exercises.**

Exercise 9.3.1. Consider Paradox 3 of Section 5. [This is usually called the "Berry Paradox," although likely due to Russell; feel free to look it up somewhere.] The following exercise is giving a simpler version of this "paradox."

9.3.1(a) Let W be the four element set

$$W = \{\texttt{one}, \texttt{two}, \texttt{plus}, \texttt{times}\}.$$

Ascribe a "meaning" to each sentence with words from W (i.e., each string over the alphabet W) in the usual way of evaluating expressions, so that

`one plus two times two` means $1 + 2 \times 2 = 5$,

`plus times two plus` is meaningless,

and each sentence either "means" some positive integer or is "meaningless." Show that every positive integer is the "meaning" of some sentence with words from W.

9.3.1(b) Show, more precisely, that there is a constant, C, such that any positive integer, n, can be described by a W-sentence of at most $1 + C(\log_2 n)^2$ words.

9.3.1(c) Consider the five element set

$$U = W \cup \{\texttt{moo}\}$$

with the following meaning for `moo`:

(a) if it appears anywhere after the first word of a sentence, then the sentence is meaningless,
(b) if it appears only once and at the beginning of a sentence, then we evaluate the rest of the sentence (as usual), and
   (i) if the rest evaluates to the integer k, then the sentence means "the smallest positive integer not described by a sentence of k words or fewer," and
   (ii) if the rest evaluates to meaningless, then the sentence is meaningless.

For example, "`moo moo`" and "`moo plus times two`" are meaningless, and "`moo two times two`" means "the smallest positive integer not described by a sentence of four words or fewer." What is the meaning of "`moo one`"?

9.3.1(d) What seems paradoxical in trying to ascribe a meaning to "`moo two`"?

9.3.1(e) Now say we are more precise, with the following meaning for `moo`:

(a) if it appears anywhere after the first word of a sentence, then the sentence is meaningless,
(b) if it appears only once and at the beginning of a sentence, then we evaluate the rest of the sentence (as usual), and
   (i) if the rest evaluates to the integer k, AND THE LENGTH OF THE WHOLE SENTENCE IS MORE THAN k, then the sentence means "the smallest positive integer not described by a sentence of k words or fewer," and
   (ii) in any other situation, the sentence is meaningless.

Hence "`moo two`" is meaningless. What is the value of "`moo one plus one`"?

Exercise 9.3.2. Explain why the following questions can't be answered either yes (true) or no (false).

9.3.2(a) In a certain village, Chris holds accountable each person who does not hold themself accountable (and no one else). Does Chris hold themself accountable?

9.3.2(b) In a certain village, Geddy is blamed by each person who does not blame themself (and by no one else). Is Geddy blamed by themself?

9.3.2(c) In a certain village, Sandy teaches each person who does not teach themself (and no one else). Does Sandy teach themself?

Exercise 9.3.3. Say that we assume that no set should contain itself (in a particular collection of axioms about set theory that we are currently using). If so, describe C given by

$$C = \{S \mid S \text{ is a set such that } S \notin S\}.$$

Explain why C cannot be a set.

9.4. **Exercises: More on Cantor's Theorem, Countable Sets.**

Exercise 9.4.1. Let $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$, i.e.,

$$\mathbb{N}^2 = \{(n_1, n_2) \mid n_1, n_2 \in \mathbb{N}\}.$$

(See Chapter 0 of [Sip].)

9.4.1(a) Show that $\mathbb{N}^2$ is countable.

9.4.1(b) Show that $\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is countable.

9.4.1(c) For any set, S, we define S^* as the union

$$S^* \overset{\text{def}}{=} S^0 \cup S^1 \cup S^2 \cup \dots$$

(which generalizes the definition of Σ^* when Σ is an alphabet). Show that $\mathbb{N}^*$ is countable.

Exercise 9.4.2. Let $C_1, C_2, \dots$ be a sequence of countably infinite sets. Is $C_1 \cup C_2 \cup \dots$ countably infinite? **Justify your answer.** (You get no credit for answering "yes" or "no" without explanation.)

Exercise 9.4.3. Recall that for $n \in \mathbb{N}$, $[n]$ denotes $\{1, 2, \dots, n\}$ (which is therefore an alphabet). The point of this exercise is to show that there are very simple surjections $[n]^* \to [m]^*$ for any $n, m \in \mathbb{N}$; it is less simple to describe bijections $[n]^* \to [m]^*$ (although not extremely difficult to do so).

9.4.3(a) Is $[2]^*$ countably infinite? Justify your answer.

9.4.3(b) Describe a simple bijection $f : [2]^* \to [4]^*$ (i.e., find a bijection that does not rely on bijections from these sets to $\mathbb{N}$).

9.4.3(c) Describe a simple bijection $f : [2]^* \to [8]^*$.

9.4.3(d) Describe a simple bijection $f : [4]^* \to [8]^*$, based on your answers to (a) and (b).

9.4.3(e) Describe a surjection $[8] \to [7]$, and use it to give a surjection $[8]^* \to [7]^*$.

9.4.3(f) Describe a surjection $[7] \to [2]$, and use it to describe a surjection $[7]^* \to [2]^*$.

9.4.3(g) Using parts (c) and (f), describe a simple surjection $[7]^* \to [8]^*$.

Exercise 9.4.4. A real number, x, is *algebraic* if it is the solution to an equation of the form

$$a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n = 0$$

where $a_0, \ldots, a_n \in \mathbb{N}$ for some $n \in \mathbb{N}$ (where $a_0 \neq 0$). Is the set of algebraic numbers countable? Justify your answer. [Hint: You may use the fact that a polynomial of degree n has at most n roots.]

Exercise 9.4.5. Which of the following sets are countably infinite? Justify your anwer.

9.4.5(a) The negative rational numbers.
9.4.5(b) The real numbers in the closed interval $[1, 2]$.
9.4.5(c) The real numbers in the open interval $(1, 2)$.
9.4.5(d) The set of all functions $\Sigma^* \to \{\texttt{yes}, \texttt{no}\}$, where Σ is an alphabet.
9.4.5(e) The set of all functions $\{\texttt{yes}, \texttt{no}\} \to \Sigma^*$, where Σ is an alphabet.
9.4.5(f) The set of all functions $\Sigma^* \to \Sigma^*$, where Σ is an alphabet.

Exercise 9.4.6.

9.4.6(a) Show that the set of subsets of $\{2, 4, 6, 8, \ldots\}$ (i.e., the even positive integers) is uncountable.
9.4.6(b) Show that the set of infinite subsets of $\mathbb{N} = \{1, 2, 3, \ldots\}$ is uncountable. [Hint: use part (a).]
9.4.6(c) Show that the set of pairs (S, T) such that S, T are infinite subsets of $\mathbb{N}$ is uncountable.
9.4.6(d) Say that any string $\Sigma^*_{\mathrm{ASCII}}$ is a assigned at most one "meaning," so that each string can describe at most one function from some set to some other set. Can every bijection $S \to T$ between subsets of $\mathbb{N}$ be described by a string in $\Sigma^*_{\mathrm{ASCII}}$?
9.4.6(e) Describe a simple way to assign a "meaning" to any string in $\Sigma^*_{\mathrm{ASCII}}$ so that each bijection $S \to T$ between **finite** subsets of $\mathbb{N}$ is described by some string. (Ideally the meaning is easy to infer from the string...)

INSERT OTHER PROBLEMS HERE.

9.5. **Exercises on Universal Turing Machines: Mechanics.**

Exercise 9.5.1. Let $\Sigma = \{1, 2\}$, let $L = \Sigma^*$, and let $\Sigma_{\mathrm{TM}} = \{0, 1, \#, L, R\}$.

9.5.1(a) Give a Turning machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \texttt{blank})$ that (1) recognizes L, (2) has q_0 different from both q_{acc} and q_{rej}, and (3) has the product $|Q|\,|\Gamma|$ as small as you can subject to (1) and (2) (or reasonably small, see the rest of the question).
9.5.1(b) Giave a standardized Turing machine that recognizes the same language as the above machine.
9.5.1(c) Write the above standardized Turing machine as a word/string over Σ_{TM} as described in class.
9.5.1(d) Write the above standardized Turing machine as a word/string over Σ_{TM} and append to it the input 2121, as described in class.
9.5.1(e) Explain—without actually writing down the word/string—how to Write the above standardized Turing machine as a word/string over Σ_{TM} and append to it the input 212121, as described in class.

Exercise 9.5.2. Same problem as Exericse 9.5.1 for the language $L = \emptyset$.

Exercise 9.5.3. Same problem as Exericse 9.5.1 for the language L described by the regular expression $1(1 \cup 2)^*$.

Exercise 9.5.4. Same problem as Exericse 9.5.1 for the language L described by the regular expression $(1 \cup 2)^*2$.

Exercise 9.5.5. Is the set of standardized Turing machines countable or uncountable? Explain.

Exercise 9.5.6. Is the set/class/family/etc. of (all) Turing machines countable or something else (e.g., uncountable, so large that it isn't even a class, etc.)? Explain.

INSERT MORE EXERCISES HERE

9.6. **A Hierarchy of Acceptance, a Hierarchy of Halting.**

Exercise 9.6.1. Let $\Sigma_{\mathrm{TM}} = \{0, 1, \#, L, R\}$. Let $\pi \colon \Sigma_{\mathrm{TM}} \to [5] = \{1, \ldots, 5\}$ be an arbitrary bijection.

9.6.1(a) If $w = \sigma_1 \ldots \sigma_n \in \Sigma_{\mathrm{TM}}^*$ is a word, let

$$\pi(w) = \pi(\sigma_1) \ldots \pi(\sigma_n).$$

Does this give a bijection between elements of Σ_{TM}^* and elements of $[5]^*$? Explain.

9.6.1(b) If L is a language over Σ_{TM}, let

$$\pi(L) = \{\pi(w) \mid w \in L\}. \tag{23}$$

Does this give a bijection between laguages over Σ_{TM}^* and languages over $[5]^*$? Explain.

Exercise 9.6.2. Let $s \in \mathbb{N}$, and let $\Sigma = [s] = \{1, \ldots, s\}$.

9.6.2(a) Explain how to define a standardized 2-tape Turing machine—using the idea of a regular standardized (1-tape) Turing machine—in a way that any 2-tape Turing machine for a language over Σ has an equivalent standardized machine that returns the same result (accept, reject, loops, i.e., $\texttt{yes}, \texttt{no}, \texttt{loops}$).

9.6.2(b) Do the same for k-tapes for $k \in \mathbb{N}$ for any $k \geq 3$.

9.6.2(c) Can you define a *standardized Turing machine* that allows you to first write down a value of k and then describe a standardized k-tape machine? Explain.

9.6.2(d) Let $s' \in \mathbb{N}$, $\Sigma_{\mathrm{oracle}} = [s'] = \{1, \ldots, s'\}$, and $A \subset \Sigma^*$. Can you define a *standardized oracle Turing machine* that has access to a single oracle A, and some standardized Turing machine as in part (c)? What conventions do you have specify?

Exercise 9.6.3. Let $A \subset \Sigma^*$ be any fixed language, A, over an alphabet Σ of the form $\{1, \ldots, s\}$ for some $s \in \mathbb{N}$. Let $\mathcal{P}$ be the set of all standardized oracle Turing machines that can make an oracle query to A, standardized appropriately (one way of standardizing is given in the above exercises). Let $\mathcal{I} = \Sigma^*$.

9.6.3(a) Show that the result of running any oracle Turing machine in $\mathcal{P}$ on an input in $\mathcal{I}$ gives an expressive program-input system.

9.6.3(b) Show that this expressive program-input system has a universal program.

9.6.3(c) Conclude that this program-input has a delightful program.

9.6.3(d) Conclude that the acceptance problem in this program-input is undecidable, i.e., there is no Turing machine with oracle A that decides the acceptance problem for Turing machines with oracle A.

Exercise 9.6.4. Let $A \subset \Sigma^*$ be any fixed language, A, over an alphabet Σ of the form $\{1, \ldots, s\}$ for some $s \in \mathbb{N}$. Let us further assume that $s \geq 5$, so that we may identify $\Sigma_{\rm TM}$ with a subset of $\Sigma = [s]$, and that we have a standardization of all multitape Turing machines as described in the problems above. Let

$$B = \mathrm{ACCEPTANCE}^A = \mathrm{ACCEPTANCE}_{\text{oracle } A}.$$

9.6.4(a) Show that if M is any oracle Turing machine with an oracle call to A, and w is an input to M, then after some preprocessing one can make a single oracle call to B to determine whether or not M accepts w.

9.6.4(b) Hence conclude that if an oracle Turing machine M^A decides a language, L, then L is also decided by some oracle Turing machine $(M')^B$ (i.e., an oracle machine that calls B, rather than A).

9.6.4(c) Using $\mathrm{Decidable}_\Sigma(A)$ to denote the class of languages over Σ decidable with an oracle A Turing machine, conclude that

$$\mathrm{Decidable}(A) \subset \mathrm{Decidable}(B) = \mathrm{Decidable}\bigl(\mathrm{ACCEPTANCE}^A\bigr)$$

9.6.4(d) Explain why $B \in \mathrm{Decidable}(B)$ (immediately) and, from the above, $B \notin \mathrm{Decidable}(A)$.

9.6.4(e) Conclude that there is a hierarchy of Turing machine oracles

$$\emptyset,\ \mathrm{ACCEPTANCE},\ \mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}},\ \mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}}},\ \ldots$$

of successively more powerful oracles, in the sense that there is a sequence of strict inclusions

$$\mathrm{Decidable}(\emptyset) \subset \mathrm{Decidable}(\mathrm{ACCEPTANCE}) \subset \mathrm{Decidable}\Bigl(\mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}}\Bigr) \subset \cdots$$

Exercise 9.6.5. Same exercise as above, except with ACCEPTANCE replaced everywhere with HALT.

Exercise 9.6.6. In the sequence

$$\emptyset,\ \mathrm{ACCEPTANCE},\ \mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}},\ \mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}^{\mathrm{ACCEPTANCE}}},\ \ldots,$$

should the first term be $\emptyset$ or its complement, Σ^*? Does it really matter?

Appendix A. *Most Languages are Unrecognizable

There are a number of well-known senses that say that "most" languages are unrecognizable. If you believe that "most" elements of an uncountable set lie outside of any given countable subset, then that is enough. Otherwise here are some other ways to make sense of this statement; these require more mathematical sophistication than we typically assume in CPSC 421/501 (as of 2023). All these are based on convincing yourself that any countable subset of either $[0, 1]$ or $\mathrm{Power}(\Sigma^*)$ has 0 "measure" or "probability" in a space of positive measure.

(1) If Σ is any finite alphabet, there is a *measure*[29] on $\mathrm{Power}(\Sigma^*)$ that for any finite $S \subset \Sigma^*$ assigns the measure $1/2^{|S|}$ to the subset of $\mathrm{Power}(\Sigma^*)$ consisting of all languages containing S; moreover, it is a probability measure, assigning the measure 1 to $\mathrm{Power}(\Sigma^*)$). Any countable subset of $\mathrm{Power}(\Sigma^*)$ can be covered by a countable collection of sets whose measure is arbitrarily small, and therefore any countable set has measure 0.
(2) Build a surjective map $\mathrm{Power}(\Sigma^*) \to [0,1]$ such that each real number has at most 2 preimages; convince yourself that for this reason, any countable set in $\mathrm{Power}(\Sigma^*)$ should have zero measure.
(3) Convince yourself that any countable subset of $[0,1]$ has zero measure. If $A \subset \mathbb{N}$, we define
$$\mathrm{Density}(A,n) = \frac{|A \cap [n]|}{n}.$$
If
$$\lim_{n\to\infty} \mathrm{Density}(A,n)$$
has a limit, we call this limit the *density* of A (hence the density of odd numbers is $1/2$). To extend this idea, note that $0 \le \mathrm{Density}(A,n) \le 1$, and hence for any A we can define its density for a subsequence of n over which the limit exists; we can get such a density function defined unambiguously for all $A \subset \mathbb{N}$ by setting
$$\mathrm{Density}_{\mathfrak{U}}(A) = \lim_{\{n\}\subset\mathfrak{U}} \mathrm{Density}(A,n)$$
with the choice of an *ultrafilter*, $\mathfrak{U}$; assuming the ultrafilter is *non-principal*, it follows that this limit agrees with the limit $n \to \infty$ of $\mathrm{Density}(A,n)$ when it exists. Then the densities of a countable subset of $\mathrm{Power}(\mathbb{N})$ is some countable set in $[0,1]$. (Taking an appropriate bijection $\mathbb{Z}^n \to \mathbb{N}$ and using the same idea we see that $\mathbb{Z}^n$ is an *amenable group*.)

Appendix B. Decision Problems, Alphabets, Strings, and Languages: More Details

In this section we explain the connection between algorithms, decision problems, and some of the definitions in Chapter 0 of [Sip]. We also discuss *descriptions*, needed starting in Chapter 3 of [Sip].

B.1. Decision Problems and Languages.

B.1. Decision Problems and Languages. The term *decision problem* refers to the following type of problems:
(1) Given a natural number, $n \in \mathbb{N}$, give an algorithm to decide if n is a prime.
(2) Given a natural number, $n \in \mathbb{N}$, give an algorithm to decide if n is a perfect square.
(3) Given a natural number, $n \in \mathbb{N}$, give an algorithm to decide if n can be written as the sum of two prime numbers.
(4) Given sequence of DNA bases, i.e., a string over the alphabet $\{C, G, A, T\}$, decide if it contains the string "ACT" as a substring.

[29]in the sense of measure theory; the values of this measure are uniquely determined on the smallest σ-field (i.e., σ-algebra) containing $\mathrm{Contains}(S)$, where S varies over all finite subsets of Σ^* and where $\mathrm{Contains}(S)$ is the subset of languages over Σ that contain S. (One could extend this to the smallest σ-field containing these subsets and all the outer measure 0 subsets.)

(5) Given an ASCII string, i.e., a finite sequence of ASCII characters[30], decide if it contains the string "CPSC 421" as a substring.
(6) Given an ASCII string, decide if it contains the string "vacation" as a substring.
(7) Given an ASCII string, decide if it is a valid C program.

Roughly speaking, such problems take an *input* and say "yes" or "no"; the term *decision problem* suggests that you are looking for an *algorithm*[31] to correctly say "yes" or "no" in a finite amount of time.

To make the term *decision problem* precise, we use the following definitions.

(1) An *alphabet* is a finite set, and we refer to its elements as *symbols*.
(2) If $\mathcal{A}$ is an alphabet, a *string over* $\mathcal{A}$ is a finite sequence of elements of $\mathcal{A}$; we use $\mathcal{A}^*$ to denote the set of all finite strings over $\mathcal{A}$.
(3) If $\mathcal{A}$ is an alphabet, a *language over* $\mathcal{A}$ is a subset of $\mathcal{A}^*$.

(People often use *letter* instead of symbol, and *word* instead of string.) For example, with $\mathcal{D} = \{0, 1, \ldots, 9\}$, we use

$$\mathrm{PRIMES} = \{s \in \mathcal{D}^* \mid s \text{ represents a prime number}\}$$

and

$$\mathrm{SQUARES} = \{s \in \mathcal{D}^* \mid s \text{ represents a perfect square}\}$$

Here are examples of elements of PRIMES:

$$421, 3, 7, 31, 127, 8191, 131071, 524287, 2147483647$$

where we use the common shorthand for strings:

$$127 \text{ for } (1,2,7), \quad 131071 \text{ for } \quad (1,3,1,0,7,1), \quad \text{etc.}$$

So PRIMES is a language over the alphabet $\mathcal{D}$; when we say "the decision problem PRIMES" we refer to this language, but the connotation is that we are looking for some sort of algorithm to decide whether or not a number is prime. Here are some examples of strings over $\mathcal{D}$ that are not elements of the set PRIMES:

$$221, 320, 420, 2019.$$

B.2. **Descriptions of Natural Numbers.** From our discussion of PRIMES above, it is **not clear** if we consider 0127 to be element of PRIMES; we need to make this **more precise**. It is reasonable to interpret 0127 as the integer 127 and to specify that $0127 \in \mathrm{PRIMES}$. However, in [Sip] we will be careful to distinguish a natural number $n \in \mathbb{N}$ and

$$\langle n \rangle \quad \text{meaning the "description" of } n,$$

i.e., the string that represents n (uniquely, according to some specified convention), so the natural number 127 has a unique description as the string $(1,2,7)$, and the string $(0,1,2,7)$ is not the description of 127. With this convention, $0127 \notin \mathrm{PRIMES}$; this is also reasonable.

[Later in the course we will speak of "the description of a graph" (when studying graph algorithms), "the description of a Boolean formula" (when studying SAT, 3SAT), "the description of a Turing machine," etc. In these situtations it will be

[30]ASCII this is an alphabet of 256 letters that includes letters, digits, and common punctuation.
[31]The term *algorithm* means different things depending on the context; in CPSC 421 we will study examples of this (e.g., a DFA, NFA, deterministic Turing machine, a deterministic Turing machine with an orale A, etc.

clear why the input to an algorithm should be a description of something (as a string over some fixed alphabet) rather than the thing itself.]

If $n = \mathbb{Z}$ with $n = 127$, the symbol $\langle n\rangle$, meaning the "description of n" can refer to

(1) "1111111," when $\langle n\rangle = \langle n\rangle_2$ means the "binary representation of n" (a unique string over the alphabet $\{0,1\}$);
(2) "11201," when $\langle n\rangle = \langle n\rangle_3$ means the "base 3 representation of n" (a unique string over the alphabet $\{0,1,2\}$);
(3) "one hundred and twenty-seven," when $\langle n\rangle = \langle n\rangle_{\mathrm{English}}$ means the "English representation of n" (a unique string over the ASCII alphabet, or at least an alphabet containing the English letters, a comma, a dash, and a space);
(4) "cent vingt-sept," similarly for French, $\langle n\rangle = \langle n\rangle_{\mathrm{French}}$
(5) "wa'vatlh wejmaH Soch," similarly for Klingon[32], $\langle n\rangle = \langle n\rangle_{\mathrm{Klingon}}$;
(6) and good old "127," when $\langle n\rangle = \langle n\rangle_{10}$ means the "decimal representation of n."

Note that haven't yet specified whether or not ϵ, the empty string, is considered to be an element of PRIMES.

B.3. **More on Strings.** Chapter 0 of [Sip] uses the following notion:

(1) if $\mathcal{A}$ is an alphabet and $k \in \mathbb{Z}_{\geq 0} = \{0,1,2\ldots\}$, a *string of length k over $\mathcal{A}$* is a sequence of k elements of $\mathcal{A}$;
(2) we use $\mathcal{A}^k$ to denote the set of all strings of length k over $\mathcal{A}$;
(3) equivalently, a string of length k over $\mathcal{A}$ is a map $[k] \to \mathcal{A}$ where $[k] = \{1,\ldots,k\}$;
(4) by consequence (or convention) $\mathcal{A}^0 = \{\epsilon\}$ where ϵ, called the *empty string*, is the unique map $\emptyset \to \mathcal{A}$;
(5) a *string over* $\mathcal{A}$ is a string over $\mathcal{A}$ of some length $k \in \mathbb{Z}_{\geq 0}$;
(6) therefore $\mathcal{A}^*$ is given as
$$\mathcal{A}^* = \bigcup_{k\in\mathbb{Z}_{\geq 0}} \mathcal{A}^k = \mathcal{A}^0 \cup \mathcal{A}^1 \cup \mathcal{A}^2 \cup \cdots$$
(7) strings are sometimes called *words* in other literature;
(8) a *letter* or *symbol* of an alphabet, $\mathcal{A}$, is an element of $\mathcal{A}$.

Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, CANADA.

Email address: jf@cs.ubc.ca

URL: http://www.cs.ubc.ca/~jf

[32]Source: https://en.wikibooks.org/wiki/Klingon/Numbers.