UNCOMPUTABILITY IN CPSC421/501

JOEL FRIEDMAN

Contents

Date: Wednesday 11th October, 2023, at 13:31.
Research supported in part by an NSERC grant.

Disclaimer: The material may sketchy and/or contain errors, which I will elaborate upon and/or correct in class. For those not in CPSC 421/501: use this material at your own risk...

This version is a draft: I'm in the process of adding material and editing article as of August 2023. 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 Sections 1–6 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).

2. Some Unrecognizable Languages

In this section we give examples of "problems" that cannot be "solved by a computer program" (or "by an algorithm").

[In Appendix A (not covered in this course), we will also explain that "most" problems cannot be solved.]

This section begins by defining the notion of a *language* (i.e., a "problem") and what it means for a language to be *recognizable* (i.e., "solvable by a computer program"). We will then prove the deservedly famous *Cantor's theorem*, and use it to

produce an unrecognizable language. This language is not of direct practical interest, but its unrecognizability implies that many other languages (i.e., problems)—which are of practical interest—are also unrecognizable, or at least *undecidable*. Examples of undecidable languages include "the halting problem" and "used line of code;" examples of unrecognizable languages include "the non-halting problem" and "unused line of code."

One shortcoming of this section is that our description of "computer program" or "algorithm" (and therefore of *recognizability*) is a bit imprecise. Later in this course we will remedy this by defining *Turing machines* (in Chapter 3 of [Sip]).

2.1. **Alphabets, Strings, and Languages.** Let us review some common definitions in computer science theory (many can be found in [Sip], Chaper 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

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

A *language over* Σ refers to any subset of Σ^*.

Example 2.1. 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).

If S is any set, then Power(S) refers to the set of all subsets of S. Hence the set of *all languages over* Σ equals Power(Σ^*).

Example 2.2. Let $\Sigma_{\text{digits}} = \{0, 1, \ldots, 9\}$. 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 Σ_{digits}. More formally, if $n \in \{0, 1, 2, \ldots\}$, we use $\langle n \rangle$ to denote the associated string; hence $\langle 2023 \rangle$ is technically the string of length 4, $(2, 0, 2, 3)$, but for brevity write 2023; to avoid confusion, at times we write "the integer 2023" for the integer, and $\langle 2023 \rangle$ or "the string 2023" when referring to the string). We define

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

or, for brevity,

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

We tend to write languages in ALL CAPITAL LETTERS. Another example is

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

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

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

and

$$\text{DIV-BY-5-LEADING-ZEROS-OK}^{1}$$
$$= \{0, 5, 00, 05, 10, 15, 20, \dots, 95, 000, 005, 010, \dots, 095, 100, 105, \dots\} \subset \Sigma^*_{\text{digits}}$$

Remark 2.3. 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 $[n] = \{1, 2, \dots, n\}$, 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 $x_1, \dots, x_n$, we will fix some conventions so that $\langle f \rangle$ is a string over some alphabet that describes f (a convenient alphabet here is $\Sigma = \{x, 0, 1, \wedge, \vee, \neg, (,)\}$).

Remark 2.4. 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.

2.2. **Programs that Recognize Certain Languages.** In this section we will provide a rough framework needed to define "recognizable" languages. You should be convinced that this framework can be made precise (there are choices to make), and that the notion of "recognizable" does not depend on the choices you make.

[In Chapter 3 we will make this framework completely precise using *Turing machines*, which is a simple (but limited) model of a "computer program."]

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 $m = 0, 1, \dots, n-1$.]

Assume you have fixed conventions so that the following holds (there is some flexibility here, but most "reasonable"[2] set of conventions will work): there is a

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

[2] In class we may give examples of "unreasonable" conventions, such as that in Example 2.18.

subset of 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 halts. The program may halt for other reasons (e.g., if you divide by 0, or if it runs the entire program). A program may not halt (e.g., if it tries to execute an "infinte loop").

[It should be pretty clear how to do this, at least to readers who have had the pleasure of writing computer programs. In class I'll answer questions you have regarding the above. When we cover *Turing machines* we will have a precise version of a "program" and how to represent each as a string over some fixed alphabet.]

Say that $p \in \Sigma^*_{\text{ASCII}}$ is a valid Python program. For each $i \in \Sigma^*_{\text{ASCII}}$, one says that p *accepts* i if on input i, p eventually reaches the statement `return("yes")`. The *language recognized by* p is defined to be

$$\text{LanguageRecBy}(p) \stackrel{\text{def}}{=} \{i \in \Sigma^*_{\text{ASCII}} \mid p \text{ accepts } i\} \subset \Sigma^*_{\text{ASCII}}. \tag{1}$$

We say that $L \subset \Sigma^*_{\text{ASCII}}$ 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 2.5. The program `isPal()` above recognizes the language PALINDROME, of ASCII strings that are palindromes. Hence PALINDROME is recognizable.

Example 2.6. 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).

Now we extend the definition in (1) by defining

$$\text{LanguageRecBy}(p) \stackrel{\text{def}}{=} \emptyset, \quad \text{if } p \text{ is not a valid Python program.} \tag{2}$$

(The value $\emptyset$ is not essential to what we do, but is convenient for our exposition.)

Hence (1) and (2) give a function

$$\text{LanguageRecBy}\colon \Sigma^*_{\text{ASCII}} \to \text{Power}(\Sigma^*_{\text{ASCII}}).$$

We will now give an unrecognizable language. This follows immediately from *Cantor's theorem*.

2.3. **Cantor's Theorem and Unrecognizable Languages.**

Theorem 2.7 (Cantor's Theorem)**.** *Let S be a set, and $f\colon S \to \mathrm{Power}(S)$ be any function. Then some $T \in \mathrm{Power}(S)$ is not in the image of f. Specifically the set*

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

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 $t \notin T$. □

Example 2.8. 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 2.7 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 2.9. 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

$$\mathrm{Image}(\mathrm{ThinksIsClever}) = \Big\{\{A, B, C\}, \emptyset, \{C\}\Big\}.$$

Corollary 2.10. *Fix any set of conventions regarding Python programs that satisfies the conditions in Subsection 2.2. Then*

$$T = \{p \in \Sigma^*_{\mathrm{ASCII}} \mid p \notin \mathrm{LanguageRecBy}(p)\} \tag{3}$$

is unrecognizable.

Proof. A language is recognizable iff it is in the image of the map

$$\text{LanguageRecBy}\colon \Sigma^*_{\text{ASCII}} \to \text{Power}(\Sigma^*_{\text{ASCII}}).$$

By Cantor's theorem, T in (3) is not in the image of LanguageRecBy. □

Remark 2.11. In class on September 13, 2023, we named the language in the above corollary

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

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

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

Remark 2.12. Note that by our conventions, T above contains all strings, p, that are not valid Python programs. Say that our conventions regarding valid Python programs imply that there is some symbol $\sigma_0 \in \Sigma_{\text{ASCII}}$ 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 (trivially) recognizable language Σ^*_{ASCII} agree on 99.9999% of the ASCII strings of any sufficiently large length. In this sense, T can "mostly agree" with a recognizable language; the above theorem only asserts that no algorithm can correctly recognize T "100% of the time."

Remark 2.13. Some practical computer algorithms halt (eventually terminate) on any given input. If p is a valid Python program that halts on any given input, then one can "simulate p on input p" to check whether or not $p \in T$ with T as in Corollary 2.10.

2.4. **Undecidable and Unrecognizable Languages.** There is no obvious reason why you'd want an algorithm to recognize

$$T = \{p \in \Sigma^*_{\text{ASCII}} \mid p \notin \text{LanguageRecBy}(p)\}. \tag{4}$$

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

Assume some conventions regarding Python programs are fixed and satisfy (1)–(3) in Subsection 2.2. Furthermore assume that there is some symbol $\sigma_0 \in \Sigma_{\text{ASCII}}$ that is never used in a valid Python program; we fix such a σ_0, and call it (for reasons to become clear) the *separating symbol*. Then 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. We note that Σ^*_{ASCII} is partitioned into three sets (i.e., each element of Σ^*_{ASCII} lies in exactly one of the three sets below):

$$\text{ACCEPTANCE} = \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that accepts } i\},$$
$$\text{NON-ACCEPTANCE} = \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that does not accept } i\}.$$
$$\text{NON-PYTHON} = \Sigma^*_{\text{ASCII}} \setminus (\text{ACCEPTANCE} \cup \text{NON-ACCEPTANCE})$$

(therefore NON-PYTHON refers to strings that either don't contain a σ_0 somewhere, or are of the form $s\sigma_0 i$ where s contains no σ_0 but s is not a valid Python program). 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 2.14. Let us summarize these definitions, plus some extra ones given in class on September 13, 2023.

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
LanguageRecBy(p)	$\{i \in \Sigma^*_{\text{ASCII}} \mid p \text{ accepts } i\}$
L is *recognizable* (where $L \subset \Sigma^*_{\text{ASCII}}$)	for some p, $L = \text{LanguageRecBy}(p)$
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 $L = \text{LanguageRecBy}(p)$

We will also consider the following languages:

$$\begin{aligned}
\text{ACCEPTANCE} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that accepts } i\},\\
\text{NON-ACCEPTANCE} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that does not accept } i\},\\
\text{NON-PYTHON} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is not a valid Python program}\},\\
\text{HALTING} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that halts on } i\},\\
\text{REJECTING} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that rejects } i\},\\
\text{LOOPING} &= \{p\sigma_0 i \in \Sigma^*_{\text{ASCII}} \mid p \text{ is a valid Python program that loops on } i\}.
\end{aligned}$$

In class we will explain why:

(1) NON-PYTHON is decidable (in "polynomial time").
(2) ACCEPTANCE is recognizable by a *universal machine for Python*, i.e., one that on input $p\sigma_0 i$ checks if p is a valid program, and if so it "simulates" how p works on input i, and says `yes` if p does so on input i. [The point is that if p never halts on input i, then the universal machine does not halt on input $p\sigma_0 i$ (and hence a universal machine is not a decider), but still—by definition—it recognizes ACCEPTANCE.] For example, a "program debugger" that allows you to run through a program step-by-step (and examine the contents of its variables, etc.) is essentially a universal machine.

This will allow us to conclude that:

(1) NON-ACCEPTANCE is unrecognizable: if it were recognizable by a Python program, q, given any $p \in \Sigma^*_{\text{ASCII}}$, you could give $p\sigma_0 p$ as input to q. This gives us a Python program (or algorithm) that would recognize T in (4), which is impossible. Hence NON-ACCEPTANCE is unrecognizable. [This is an example of a "reduction," i.e., we can "reduce" T to

NON-ACCEPTANCE, i.e., we have an algorithm to solve T if we have an algorithm to solve NON-ACCEPTANCE; this yields a contradition.]

(2) ACCEPTANCE is not decidable (if it were, then you could decide the union of NON-ACCEPTANCE and NON-PYTHON, and therefore decide NON-ACCEPTANCE, hence recognize NON-ACCEPTANCE, which is impossible).

(3) 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[3]; 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[4]. Hence the complement of USED-LINE-OF-CODE (and therefore the analogously defined UNUSED-LINE-OF-CODE) is unrecognizable.

2.5. **Change of Alphabet, Generalized Cantor's Theorem.** Next imagine that your Python programs are written in ASCII, but that their inputs are strings over the alphabet

$$\Sigma_{\text{Greek}} = \{\alpha, \beta, \gamma, \ldots, \omega, A, B, \Gamma, \ldots, \Omega\}.$$

Could every language over Σ_{Greek} be recognizable, giving a surjection $L\colon \Sigma^*_{\text{ASCII}} \to \text{Power}(\Sigma^*_{\text{Greek}})$?

It turns out—which may not surprise you—that changing the alphabet does not really matter.

What if a Python program is allowed an input that is a single element of $\mathbb{N} = \{1, 2, 3, \ldots\}$ or of $\mathbb{R}$, the real numbers? What about if a program has statements involving real numbers? We will discuss such questions further in the next subsection and in Section 5.

For now, let's state an easy extension of Cantor's theorem; with a little more work (done in Section 5) shows that changing the alphabet (as above) does not matter. We need the following definitions (see also Definition 4.12 of [Sip]).

Definition 2.15. Let A, B be sets, and $f\colon A \to B$ a function (also known as a map and a morphism). We say that f is:

[3] 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.

[4] 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 recogonize USED-LINE-OF-CODE, on input $p\sigma_0\ell$, 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. 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.

(1) *injective* (also known as *an injectiion*, *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. as *a surjection*, *onto* or 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*[5] or *an isomorphism*) if it is injective and surjective.

You should convince yourself that if A, B are finite sets, then there exists (1) an injection $A \to B$ iff $|A| \leq |B|$, (2) a surjection $A \to B$ iff $|A| \geq |B|$, and (3) a bijection $A \to B$ iff $|A| = |B|$.

Say that there is a surjection $S \to S'$; then there can't exist a surjection $S' \to \text{Power}(S)$ since (we easily see that) a compositions of two surjections is a surjection, and there is no surjection $S \to \text{Power}(S)$. For finite sets, there is a surjection $S \to S'$ iff $|S'| \leq |S|$; hence if one can make sense of the statement "$|S'| \leq |S|$" for infinite sets S, S', one should be able to prove

$$|S'| \leq |S| \quad \Rightarrow \quad \text{there is no surjection } S' \to \text{Power}(S).$$

This is essentially done in Theorem 2.16 below.

We warn the reader that the notion of the "size of a set" becomes tricky with infinite sets: indeed, in Section 5 we describe a bijection from $\mathbb{N} = \{1, 2, 3, \ldots\}$ to $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, and hence we regard the two sets as having the "same size" and write $|\mathbb{N}| = |\mathbb{Z}|$ (both sets are *countably infinite*). Of course, $\mathbb{N} \subset \mathbb{Z}$ is a proper subset. For now, let's accept that there is a way to make sense of the statement $|S'| \leq |S|$, and that this implies that there is a surjection $S \to S'$[6].

Theorem 2.16 (Generalized Cantor's theorem)**.** *Let $g\colon S \to S'$ be a surjection of sets. Then for any map $f\colon S' \to \text{Power}(S)$, the image of f is not all of* Power(S)*; specifically the set*

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

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 $t \in S$ with $g(t) = t'$. Then either $t \in T$ or $t \notin T$; in both cases we easily get a contradiction; we leave the details as an exercise (Exercise 7.2.9). □

Example 2.17. Say that:

(1) student A has not seen the movie "Oppenheimer;"
(2) student B has seen the 2023 movie "Barbie" and has not seen the movie "2001: A Space Odyssey;"
(3) student C has seen the movie "Encounters at the End of the World."

We now create a subset of movies that is not the subset of movies seen by any of A,B,C: the above data gives us a surjection:

$$g\colon \{\text{Oppenheimer}, \text{Barbie}, 2001, \text{Encounters}\} \to \{A, B, C\},$$

[5] We warn the reader that a *correspondence from A to B* (without writing "one-to-one") is often defined to be a morphism $C \to A \times B$.

[6] One usually defines $|S'| \leq |S|$ to mean that there exists an injection $f\colon S' \to S$; (in typical set theories one uses) this implies that is a surjection $g\colon S \to S'$, taking g to be the inverse of f on the image of f, and taking g outside of the image of f to be, say, some fixed value. If you try to prove the converse, that given a surjection $g\colon S \to S'$ there is an injection $f\colon S' \to S$, a simple way to do this is to choose for each $s' \in S'$ an element $s \in S$ such that $g(s) = s'$: note that this proof is OK only if you are assume the Axiom of Choice.

i.e., $g\colon S \to S'$ where

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

We let

$$\begin{aligned} T &= \{s \in S \mid s \text{ has not been seen by } g(s)\} \\ &= \{\text{Oppenheimer}, 2001\} \end{aligned}$$

and consider a (possibly imaginary) student, X, who has seen Oppenheimer and 2001 but has not seen Barbie and Encounters. We see that:

(1) X cannot equal A, since X has seen Oppenheimer, but A has not;
(2) X cannot equal B, since X has not seen Barbie but B has; another way to see that X cannot equal B is that X has seen 2001, but B has not;
(3) X cannot equal C, since X has not seen Encounters but C has.

In Section 5 we will see that for alphabet, Σ, the set Σ^* is *countably infinite*; it will follow that even if two alphabets, Σ_1, Σ_2, are of different size, nonetheless Σ_1^* and Σ_2^* are "of the same size," meaning that there is a bijection $\Sigma_1^* \to \Sigma_2^*$ (and hence a surjection $\Sigma_1^* \to \Sigma_2^*$). Hence Theorem 2.16 implies that there is no surjection $\Sigma_1^* \to \text{Power}(\Sigma_2^*)$. Hence the choice of alphabet makes no difference.

2.6. *Some Subtle Issues.

Example 2.18. 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 $L \subset \text{Power}(\mathbb{N})$ 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 2.19. 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[7]: 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].

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

Example 2.20. 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.)

3. 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.
(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[8] 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 2.7, that takes a map $f\colon S \to \text{Power}(S)$ 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

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

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.

4. 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 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."

5. More on Cantor's Theorem: Countability, Yes/No Systems

In this section we make some more remarks regarding Section 2.

In Subsection 5.1, as promised at the end of Subsection 2.5, we will show that if Σ_1, Σ_2 are any two alphabets—possibly of different sizes—the sets Σ_1^* and Σ_2^* are "of the same size," i.e., there is a bijection between them. We conceptualize this by showing that both are *countably infinite*. Then generalized Cantor's theorem implies that there is no surjection $\Sigma^*_{\text{ASCII}} \to \text{Power}(\Sigma^*_{\text{Greek}})$, and no surjection $\Sigma_1^* \to \text{Power}(\Sigma_2^*)$ for any alphabets Σ_1, Σ_2.

In Subsection 5.2 we will explain why Cantor's theorem and its generalization are often described as a type of "diagonalization argument," by drawing certain "yes/no tables."

In Subsection 5.3 we introduce the framework of "yes/no systems," as a first step to understanding how the yes/no tables of Subsection 5.2 are related to the undecidability and unrecognizability. In Section 6 we will enhance these "yes/no systems" to "yes/no/loops systems" which will allow us to explain the undecidability results in Section 4.2 of [Sip] and Section 2 of this article in a very general

context that includes Python programs, and oracle Turning machines (or oracle Python programs).

5.1. **Countably Infinite Sets.**

Definition 5.1. 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 5.2. 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 5.3. 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 5.4. 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 5.5. 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 2.16, 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;
(6) Cantor's theorem implies that there is no bijection from a countable set to an uncountable set;

(7) 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.

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 5.6. 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 5.7. If $S \subset \mathbb{N}$ and there is no bijection $S \to \mathbb{N}$, then we will show that S is finite. Say that $S \subset \text{Power}(\mathbb{N})$ and that there is no bijection $S \to \text{Power}(\mathbb{N})$, is there necessarily true that S is countable? Is the answer obvious?[9].

5.2. **Cantor's Theorem, Diagonalization, and `yes`/`no` Tables.** Let us descirbe Example 2.9 in the following table:

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:

Does X think that Y is clever?	$Y = A$	$Y = B$	$Y = C$
$X = A$	`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 this way Cantor's theorem is an example of *diagonalization*; 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.

However, you don't necessarily need a diagonal: consider the diagram:

Does X think that Y is clever?	$Y = A$	$Y = B$	$Y = C$
$X = A$		`yes`	
$X = B$			`no`
$X = C$	`no`		

In this case we have chosen one entry in each row so that no two entries are in the same column; by choosing T to be the s such that column s has a `no`, we get $T = \{A, C\}$, and we see that T is also not in the image of the function

$$\text{ThinksIsClever}\colon \{A,B,C\} \to \text{Power}\big(\{A,B,C\}\big)$$

because:

[9] 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) T cannot equal ThinksIsClever(A), since ThinksIsClever(A) contains B, and T does not;
(2) T cannot equal ThinksIsClever(B), since ThinksIsClever(B) does not contain C, and T does;
(3) T cannot equal ThinksIsClever(C), since ThinksIsClever(C) does not contain A, and T does.

Hence, for any function $f\colon \{A,B,C\} \to \text{Power}(A,B,C)$ that looks like the above diagram, i.e., such that

(5)

Is $Y \in f(X)$?	$Y = A$	$Y = B$	$Y = C$
$X = A$	*anything*	`yes`	*anything*
$X = B$	*anything*	*anything*	`no`
$X = C$	`no`	*anything*	*anything*

it follows that $T = \{A, C\}$ in not in the image of f.

We can conceptualize the example above as a special case of our generalized Cantor's theorem (Theorem 2.16): indeed, let $S = S' = \{A, B, C\}$, and let $g\colon S \to S'$ be the surjection (which is also a bijection):

$$g(A) = B,\ g(B) = C,\ g(C) = A.$$

Then (5) is equivalent to:

Is $Y \in f(g(X))$?	$Y = A$	$Y = B$	$Y = C$
$X = A$	`no`	*anything*	*anything*
$X = B$	*anything*	`yes`	*anything*
$X = C$	*anything*	*anything*	`no`

It may be clearer to consider a case of generalized Cantor's theorem (Theorem 2.16) where $S \neq S'$.

Example 5.8. 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

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

the above data gives us a `yes`/`no` table: $S' \to \text{Power}(S)$

Did s' (co)write s?	$s =$ Dispossessed	$s =$ Lathe	$s =$ Left	$s =$ Leviathan
$s' =$ Ursula	`yes`	`yes`	`yes`	`no`
$s' =$ Daniel	`no`	`no`	`no`	`yes`
$s' =$ Ty	`no`	`no`	`no`	`yes`

However, even with only the partial information:

Did s' (co)write s?	$s =$ Dispossessed	$s =$ Lathe	$s =$ Left	$s =$ Leviathan
$s' =$ Ursula			`yes`	
$s' =$ Daniel		`no`		`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 $g(s)$ (co)write s ? " where we set

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

notice that g is surjective. Hence the set of s such that $g(s)$ did not (co)write s is

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

and no author (co)wrote exactly this subset T of books.

Example 5.9. Let S, S' be as in Example 5.8, and say that we know the following partial information on which authors (co)wrote the following books:

Did s' (co)write s?	$s =$ Dispossessed	$s =$ Lathe	$s =$ Left	$s =$ Leviathan
$s' =$ Ursula			`yes`	
$s' =$ Daniel	`no`	`no`		`yes`
$s' =$ Ty				

This table answers the question "did $g(s)$ (co)write s" where

$$g(\text{Dispossessed}) = \text{Daniel},\ g(\text{Lathe}) = \text{Daniel},\ g(\text{Left}) = \text{Ursula},\ g(\text{Leviathan}) = \text{Daniel};$$

but g is not a surjection. Hence we cannot use Theorem 2.16 to infer a subset of books that isn't (co)written by any author. Of course, the row for Ty is blank, which is another reason that we can't infer such a subset. Since there are 16 such subsets, and only 3 authors, it is clear that there are at least 13 such subsets (and if Ty and Daniel have identical authorships, and Ursula differs with Ty and Daniel, then there are exactly 14 such subsets).

5.3. **Generalized Cantor's Theorem as Abstract Computation: Yes/No Systems.** We can rewrite generalized Cantor's theorem (Theorem 2.16) to look more like the theory of algorithms.

Definition 5.10. By a *yes/no system* we mean a 3-tuple $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R)$ where $\mathcal{P}$ is a set called the *set of programs*, $\mathcal{I}$ is a set called the *set of inputs*, and $R\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{no}, \texttt{yes}\}$ is a function, called the *result function*. For each $p \in \mathcal{P}$, we define

$$\text{Language}(p) = \text{Language}_{\mathcal{S}}(p) = \{i \in \mathcal{I} \mid \text{Result}(p, i) = \texttt{yes}\} \subset \mathcal{I}$$

and we call $\text{Language}(p)$ the *set of inputs recognized by* p. We say that a set $L \subset \mathcal{I}$ is *recoginizable* by M if it is recognized by some $p \in \mathcal{P}$.

Example 5.11. Let $\mathcal{P} = \mathcal{I} = \Sigma^*_{\text{ASCII}}$ and let $\mathcal{P} \subset \Sigma^*_{\text{ASCII}}$ be the subset of valid Python programs (in some agreed upon set of conventions). Output$\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{no}, \texttt{yes}\}$ be given as follow:

(1) if p is a valid Python program, then $\text{Answer}(p, i)$ is `yes` if p accepts i, and otherwise `no`[10]; and
(2) if p is not a valid Python program, then then $\text{Answer}(p, i) =$ `no`.

Then $\text{Language}_{\text{Output}}$ equals the function LanguageRecBy desribed in Subsection 2.2.

[10]We caution the reader that if p is a valid Python program, then $\text{Answer}(p, i) =$ `no` simply means that p does not stop after a finite number of steps and print `yes`; hence if p on input i never halts, then by convention $\text{Answer}(p, i) =$ `no`.

Example 5.12. Every `yes`/`no` table of the previous subsection can be viewed as a yes/no system $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R)$. For example, Example 5.8 fits this formalism, where

$$\mathcal{I} = S = \{\text{Dispossessed}, \text{Lathe}, \text{Left}, \text{Leviathan}\},$$

$$\mathcal{P} = S' = \{\text{Ursula}, \text{Daniel}, \text{Ty}\},$$

and $R(p, i) = \texttt{yes}$ iff i was (co)written by p. All other examples of the previous subsection have $\mathcal{P} = \mathcal{I} = \{A, B, C\}$, and $R(p, i) = \texttt{yes}$ iff p thinks that i is clever.

The following is generalized Cantor's theorem, stated in terms of yes/no systems.

Theorem 5.13. *Let $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R)$ be a yes/no system, and let $g\colon \mathcal{I} \to \mathcal{P}$ be a surjective function. Then the set*

$$T = \{i \in \mathcal{I} \mid \text{Result}(g(i), i) = \texttt{no}\}$$

is not recognizable.

The special case where $\mathcal{I} = \mathcal{P}$ and g is the identity function is Cantor's theorem, where S there equals $\mathcal{I} = \mathcal{P}$.

Proof. Assume to the contrary that $\text{Language}_{\text{Result}}(p) = T$ for some $p \in \mathcal{P}$. Since g is surjective, we can choose an i with $g(i) = p$. Then either $R(g(i), i)$ equals `yes` or `no`; either way we easily derive a contradiction (the details are an exercise). □

6. Undecidability, Acceptance, Halting, and Delightful Programs

In Section 2, we proved that in the context of Python programs,

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

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 of that ACCEPTANCE is undecidably is different—at least in spirit—from the usual proof in that:

(1) we define a *delightful program* to be any program that recognizes ACCEPTANCE (such programs exist in many contexts, including for Turing machines, Python programs, oracle Turing machines, 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"; and
(3) as an immediate consequence, the ACCEPTANCE problem is undecidable.

Since most of the work in (1)–(3) above is part (2), we spend most of our time proving a "true result" about certain programs—*delightful programs*—that do exist, and derive the undecidability of the acceptance problem as an immediate consequence[11].

[11] 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.

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 *yes/no systems* of Definition 5.10 with some additional finer structure (e.g., the `no` of a yes/no system becomes either a `no` or a `loops` in a yes/no/loops system, but *recognizability* means the same thing in both systems).

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].

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

Definition 6.1. By a *yes/no/loops system* we mean a triple $\mathcal{S} = (\mathcal{P}, \mathcal{I}, R, \text{EncodeP}, \text{EncodeBoth})$ 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 EncodeP(p), and $\langle p, i \rangle$ for 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 6.2. Fix some conventions regarding valid Python programs, such that no Python program contains a symbol $\sigma_0 \in \Sigma_{\text{ASCII}}$. Then we may take $\mathcal{P} = \mathcal{I} = \Sigma^*_{\text{ASCII}}$, where $\langle p \rangle$ is p itself, and set $\langle p, i \rangle = p\sigma_0 i$.

Example 6.3. Say that in the above example all symbols $\sigma_0 \in \Sigma_{\text{ASCII}}$ 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 $\langle p, i \rangle \stackrel{\text{def}}{=} p\sigma_0 i$ may no longer be an injection). In this case, for any string $s = \sigma_1 \dots \sigma_k$ of length k, define Duplex(s) to be the string of length $2k$ given by

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

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

Example 6.4. 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 6.5. 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 6.6. 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

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

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

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

and

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

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

$$\forall m \in \mathcal{P}, \quad R\big(q, \langle m\rangle\big) = \neg R\Big(p, \big\langle m, \langle m\rangle\big\rangle\Big).$$

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

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 6.8. *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

$$R(d, \langle d \rangle) = \neg R(h, \langle d, \langle d \rangle \rangle). \tag{6}$$

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

(2) follows from (1) and (6). □

Corollary 6.9. *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,* $\text{ACCEPTANCE}_{\mathcal{S}}$ *is undecidable.*

6.2. **Delightful Turing Machines and Undecidability.**

Definition 6.10. We say that a Turing machine is *delightful* if it recognizes the language

$$\text{A}_{\text{TM}} = \text{ACCEPTANCE}_{\text{TM}} = \{\langle M, i \rangle \mid M \text{ accepts } i\}.$$

For example, a universal Turning 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 $\text{ACCEPTANCE}_{\text{TM}}$, 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 6.11. 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 6.12. *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 6.13. *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 6.12. □

7. EXERCISES

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

Subsections 7.5 and 7.6 will not be covered until we discuss Turing machines in Chapter 3 of [Sip].

7.1. 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 $\mathbb{N}$.

7.2. Exercises for Section 2: Cantor's Theorem, Yes/No Tables, Etc.

Exercise 7.2.1. Let $S = \{1, 2, 3\}$ and $f\colon S \to \text{Power}(S)$ 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 7.2.2. Let $S = \{1, 2, 3\}$ and $f\colon S \to \text{Power}(S)$ 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 7.2.3. Let $S = \{1, 2, 3\}$ and $f\colon S \to \text{Power}(S)$ satisfy $f(1) = \{1, 2\}$. Let $T = \{s \mid s \notin f(s)\}$.

7.2.3(a) Show that $T \neq \{1, 2\}$.

7.2.3(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 7.2.4. Let $f\colon \mathbb{N} \to \text{Power}(\mathbb{N})$ 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 7.2.5. 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

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

7.2.5(a) What does the above information tell you about f ?

7.2.5(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 $T \neq f(C)$?

7.2.5(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 7.2.6. 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 7.2.7. 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 7.2.8. 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 7.2.9. Complete the proof of Theorem 2.16.

Exercise 7.2.10. 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 [12].

7.2.10(a) What is T? In other words, list the elements between braces ($\{, \}$).

7.2.10(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 7.2.11. Same as Exercise 7.2.10, 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 7.2.12. Same as Exercise 7.2.10, with the modification that no one loves anyone.

Exercise 7.2.13. Consider the setting in Exercise 7.2.10, with the modification that everyone loves everyone.

7.2.13(a) What is T?

7.2.13(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 7.2.14. 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;
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.

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

7.2.14(a) What is S?

7.2.14(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.

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

7.2.14(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 7.2.15. Consider the same situation as Exercise 7.2.14. 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

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

7.2.15(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.

7.2.15(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.

7.2.15(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.

7.2.15(d) If $f\colon V \to V$ were any other function—not necessarily a bijection—would part (c) still be true?

Exercise 7.2.16. 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 7.2.17. 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 7.2.18. 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 7.2.19. Which of the following maps are injections (i.e., one-to-one), and which are surjections (i.e., onto)? Briefly justify your answer.

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

Exercise 7.2.20. 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.

Exercise 7.2.21. Let $S = \{a, b, c\}$ and let $f\colon S \to \text{Power}(S)$ any function such that

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

7.2.21(a) Explain why $f(a)$ cannot be all of S.
7.2.21(b) Explain why none of $f(a), f(b), f(c)$ equal S.
7.2.21(c) What is the set

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

Exercise 7.2.22. Let $S = \{a, b, c\}$ and let $f\colon S \to \text{Power}(S)$ any function such that

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

7.2.22(a) Explain why $f(b)$ cannot equal $\{a, c\}$.
7.2.22(b) Explain why none of $f(a), f(c)$ equal $\{a, c\}$.
7.2.22(c) What is the set

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

Exercise 7.2.23. Let

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

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 $f\colon S' \to \text{Power}(S)$.

7.2.23(a) What can you assert about $f(A)$? What do you NOT know about $f(A)$?
7.2.23(b) Give a surjection $g\colon S \to S'$ such that for each $x \in S$ you can answer the question "is x in $f(g(x))$?"
7.2.23(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 7.2.24. Let

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

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 7.2.25. 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.

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

Exercise 7.2.26. 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 Σ_{ASCII} such that no valid Python program contains σ_0 (in class we imagined this to be σ_0 equal to $\langle\text{BELL}\rangle$, the "bell symbol" in Σ_{ASCII}). **Justify your answer (no points are given for an answer without explanation).**

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

Exercise 7.2.27. [13] Let $i_1, i_2, \ldots$ be a sequence elements of Σ^*_{ASCII} such that each element of Σ^*_{ASCII} appears exactly once in this sequence.[14] 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.:

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

[14] 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 Σ_{ASCII}, $i_{130}, \ldots, i_{1+128+128^2}$ are the elements of Σ^2_{ASCII}, 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, 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.

7.2.27(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 $O(1)$).

7.2.27(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**

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

7.2.27(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 ?

7.2.27(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 7.2.28. Continuing with the setup and notation as in the previous problem:

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

7.2.28(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.]

7.3. **Paradox Exercises.**

Exercise 7.3.1. Consider Paradox 3 of Section 3. [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."

7.3.1(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

$$\texttt{one plus two times two} \quad \text{means} \quad 1 + 2 \times 2 = 5,$$

`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.

7.3.1(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.

7.3.1(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`"?

7.3.1(d) What seems paradoxical in trying to ascribe a meaning to "`moo two`"? What do you think is the "best" interpretation of "`moo two`", and why won't this completely satisfy your notion of the word "describe"? [This question has a few correct answers, none particularly better than the others. If this last question seems strange or wrong, make up your own version of this question and answer it.]

Exercise 7.3.2. Explain why the following questions can't be answered either yes (true) or no (false).

7.3.2(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?

7.3.2(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?

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

Exercise 7.3.3. 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.

7.4. Exercises for Section 5: Countable Sets, Yes/No Tables, Etc.

Exercise 7.4.1. 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].)

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

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

Exercise 7.4.2. Let $C_1, C_2, \ldots$ be a sequence of countably infinite sets. Is $C_1 \cup C_2 \cup \ldots$ countably infinite? **Justify your answer.** (You get no credit for answering "yes" or "no" without explanation.)

Exercise 7.4.3. Recall that for $n \in \mathbb{N}$, $[n]$ denotes $\{1, 2, \ldots, n\}$ (which is therefore an alphabet).

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

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

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

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

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

7.4.3(f) Using parts (b) and (e), describe a simple surjection $[2]^* \to [7]^*$.

Exercise 7.4.4. A real number, x, is *algebraic* if it is the solution to an equation of the form

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n = 0$$

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

7.4.5(a) The negative rational numbers.

7.4.5(b) The real numbers in the closed interval $[1, 2]$.

7.4.5(c) The real numbers in the open interval $(1, 2)$.

7.4.5(d) The set of all functions $\Sigma^* \to \{\texttt{yes}, \texttt{no}\}$, where Σ is an alphabet.

7.4.5(e) The set of all functions $\{\texttt{yes}, \texttt{no}\} \to \Sigma^*$, where Σ is an alphabet.

7.4.5(f) The set of all functions $\Sigma^* \to \Sigma^*$, where Σ is an alphabet.

Exercise 7.4.6. 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.

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

7.4.6(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?"

7.4.6(c) Use g and Theorem 2.16 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 $F \setminus T$).

Exercise 7.4.7. Consider the setting of Exercise 7.2.10. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 7.4.8. Consider the setting of Exercise 7.2.11. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 7.4.9. Consider the setting of Exercise 7.2.12. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

Exercise 7.4.10. Consider the setting of Exercise 7.2.13. Draw a yes/no table of who loves whom, and explain why the set T is said to be constructed "by diagonalization."

INSERT OTHER PROBLEMS HERE.

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

Exercise 7.5.1. Let $\Sigma = \{1, 2\}$, let $L = \Sigma^*$, and let $\Sigma_{\text{TM}} = \{0, 1, \#, L, R\}$.

7.5.1(a) Give a Turning machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \texttt{blank})$ 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).

7.5.1(b) Giave a standardized Turing machine that recognizes the same language as the above machine.

7.5.1(c) Write the above standardized Turing machine as a word/string over Σ_{TM} as described in class.

7.5.1(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.

7.5.1(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 7.5.2. Same problem as Exericse 7.5.1 for the language $L = \emptyset$.

Exercise 7.5.3. Same problem as Exericse 7.5.1 for the language L described by the regular expression $1(1 \cup 2)^*$.

Exercise 7.5.4. Same problem as Exericse 7.5.1 for the language L described by the regular expression $(1 \cup 2)^*2$.

Exercise 7.5.5. Is the set of standardized Turing machines countable or uncountable? Explain.

Exercise 7.5.6. 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

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

Exercise 7.6.1. Let $\Sigma_{\text{TM}} = \{0, 1, \#, L, R\}$. Let $\pi\colon \Sigma_{\text{TM}} \to [5] = \{1, \ldots, 5\}$ be an arbitrary bijection.

7.6.1(a) If $w = \sigma_1 \dots \sigma_n \in \Sigma_{\text{TM}}^*$ is a word, let

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

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

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

$$\pi(L) = \{\pi(w) \mid w \in L\}. \tag{7}$$

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

Exercise 7.6.2. Let $s \in \mathbb{N}$, and let $\Sigma = [s] = \{1, \dots, s\}$.

7.6.2(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., `yes`, `no`, `loops`).

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

7.6.2(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.

7.6.2(d) Let $s' \in \mathbb{N}$, $\Sigma_{\text{oracle}} = [s'] = \{1, \dots, s'\}$, 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 7.6.3. Let $A \subset \Sigma^*$ be any fixed language, A, over an alphabet Σ of the form $\{1, \dots, s\}$ 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 $\mathcal{I} = \Sigma^*$.

7.6.3(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.

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

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

7.6.3(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 7.6.4. Let $A \subset \Sigma^*$ be any fixed language, A, over an alphabet Σ of the form $\{1, \dots, s\}$ for some $s \in \mathbb{N}$. Let us further assume that $s \geq 5$, so that we may identify Σ_{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

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

7.6.4(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.

7.6.4(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).

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

$$\text{Decidable}(A) \subset \text{Decidable}(B) = \text{Decidable}\big(\text{ACCEPTANCE}^A\big)$$

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

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

$$\emptyset,\ \text{ACCEPTANCE},\ \text{ACCEPTANCE}^{\text{ACCEPTANCE}},\ \text{ACCEPTANCE}^{\text{ACCEPTANCE}^{\text{ACCEPTANCE}}},\ \dots$$

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

$$\text{Decidable}(\emptyset) \subset \text{Decidable}(\text{ACCEPTANCE}) \subset \text{Decidable}\Big(\text{ACCEPTANCE}^{\text{ACCEPTANCE}}\Big) \subset \cdots$$

Exercise 7.6.5. Same exercise as above, except with ACCEPTANCE replaced everywhere with HALT.

Exercise 7.6.6. In the sequence

$$\emptyset,\ \text{ACCEPTANCE},\ \text{ACCEPTANCE}^{\text{ACCEPTANCE}},\ \text{ACCEPTANCE}^{\text{ACCEPTANCE}^{\text{ACCEPTANCE}}},\ \dots,$$

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 $\text{Power}(\Sigma^*)$ has 0 "measure" or "probability" in a space of positive measure.

(1) If Σ is any finite alphabet, there is a *measure*[15] on $\text{Power}(\Sigma^*)$ that for any finite $S \subset \Sigma^*$ assigns the measure $1/2^{|S|}$ to the subset of $\text{Power}(\Sigma^*)$ consisting of all languages containing S; moreover, it is a probability measure, assigning the measure 1 to $\text{Power}(\Sigma^*)$). Any countable subset of $\text{Power}(\Sigma^*)$ 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 $\text{Power}(\Sigma^*) \to [0,1]$ such that each real number has at most 2 preimages; convince yourself that for this reason, any countable set in $\text{Power}(\Sigma^*)$ should have zero measure.

(3) Convince yourself that any countable subset of $[0,1]$ has zero measure. If $A \subset \mathbb{N}$, we define

$$\text{Density}(A,n) = \frac{|A \cap [n]|}{n}.$$

If

$$\lim_{n\to\infty} \text{Density}(A,n)$$

[15] in the sense of measure theory; the values of this measure are uniquely determined on the smallest σ-field (i.e., σ-algebra) containing $\text{Contains}(S)$, where S varies over all finite subsets of Σ^* and where $\text{Contains}(S)$ 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.)

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 $0 \leq \text{Density}(A, n) \leq 1$, 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

$$\text{Density}_{\mathfrak{U}}(A) = \lim_{\{n\} \subset \mathfrak{U}} \text{Density}(A, n)$$

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 $\text{Density}(A, n)$ when it exists. Then the densities of a countable subset of $\text{Power}(\mathbb{N})$ 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.** 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.
(5) Given an ASCII string, i.e., a finite sequence of ASCII characters[16], 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*[17] 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 $\mathcal{A}^*$.

[16] ASCII this is an alphabet of 256 letters that includes letters, digits, and common punctuation.

[17] 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.

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

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

and

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

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 $0127 \in \text{PRIMES}$. 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, $0127 \notin$ 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 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 $\langle n \rangle = \langle n \rangle_{\text{English}}$ 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, $\langle n \rangle = \langle n \rangle_{\text{French}}$
(5) "wa'vatlh wejmaH Soch," similarly for Klingon[18], $\langle n \rangle = \langle n \rangle_{\text{Klingon}}$;
(6) and good old "127," when $\langle n \rangle = \langle n \rangle_{10}$ means the "decimal representation of n."

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

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.

E-mail address: jf@cs.ubc.ca

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