UNCOMPUTABILITY OR RUINING THE SUPRISES IN CPSC421/501

JOEL FRIEDMAN

Contents

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...

The reference [Sip] is to the course textbook, *Introduction to the Theory of Computation* by Michael Sipser, 3rd Edition.

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., $n \log_2 n + 3n + 5 = n \log_2 n + O(n)$).

In Appendix A and B we include some material handed out in the Fall 2020 version of this course. These appendices overlap with the main part of this article, but parts of the appendices give a more leisurely (and possibly more tedious) review of of strings/words, languages, and uncountable sets.

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 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 this version (2021).

1. The Main Goals of This Article

One main goal of this article is to introduce some material that is typical of the level of difficulty of CPSC 421/501. The problem with standard introductory textbooks to computer science theory, including the one we use, is that the more difficult material typically occurs well after the first two weeks. This gives a misleading idea of what is expected of students.

Another main goal of this article is to totally ruin all the main surprises in this course. The most interesting parts of this course are:

(1) "self-referencing" leads to wonderful "paradoxes," which can give interesting theorems (e.g., the Halting Problem is undecidable);

(2) certain discrete math problems can "simulate" computers, so that if you could solve these problems you could solve any problem solvable by such computers (NP-completeness).

The description of these two parts may seem a little vague. By the end of the course you should be able to state precise theorems regarding these ideas, and have tools to know when (in practice) you are encountering such problems.

For the first few weeks of CPSC 421/501, we will study computability in an abstract setting, and describe why we usually get problems that are not "computable" in some sense. We will then try to make this precise but concrete, and we will see why Turning machines are popular for doing this. We shall also see many related problems and "paradoxes" in other fields. Sipser's text [Sip] (or almost any introductory theory text) will explain the Halting Problem in more detail, and we shall later return to the Halting Problem and cover it rigorously.

2. The Main Goal of CPSC 421/501

The main point of CPCS 421/501 is to address the following questions:

(1) What is meant by the problem "P versus NP," and what are its ramifications?
(2) How might one go about resolving "P versus NP" (e.g., circuit complexity, ideas from regular languages)?
(3) Which approaches to solving "P versus NP" will not work (e.g., the Baker-Gil-Soloway Theorem, the Razborov-Rudich Theorem).

In brief, the problem "P versus NP" is generally considered to be one of the most important questions in the field(s) of algorithms and computer science theory; its study and that of related problems has given rise to many interesting developments in these fields. This problem was first precisely stated in the 1970's, with partial formulations likely going back to the 1950's, and certain algorithmic questions arguably going back centuries if not millennia. **If any student(s) solves this problem this term, I will request that the university give an exceptional grade of 101 out of 100 to this (these) student(s).**

3. Some "Paradoxes"

The two main results of this course are perhaps (1) the unsolvabilitiy of the halting problem, and (2) NP-completeness. Both results are 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[1] to destroy the leader of an army of robots. This leads to a paradox: if the speaker is telling the truth, then he is lying ("I am lying"), but if he is lying, then he is lying that he is lying, meaning he is 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.

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 precise terms.

In this course we take 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 it free of contradictions.
(3) The paradox doesn't go away, but only if certain things are true. Therefore (assuming your setting is free of contradictions) you have a theorem that one of these certain things must be false.

As examples, the "smallest number" paradox gives result (1), i.e., the paradox goes away when things are stated precisely; the set theory paradox gives result (2), i.e., it does not go away, and set theory had to be rewritten so as not to allow the formation of concepts like, "the set of all sets that blah blah blah." The halting problem is an example of result (3), and gives us a theorem (woo-hoo!) about a problem that cannot be solved by an algorithm or computer.

In order to show the Halting Problem is undecidable, we need to make our setting precise.

5. The Pigeon Hole Principle and Professor-Ice Cream Systems

In this section we will the classical *pigeon hole* principle as it applies to pigeons and "pigeon holes" (e.g., bird houses or bird sanctuaries). We will then apply these principles when the pigeons are replaced by professors, and each prof declares whether or not they like certain flavours of ice cream. The bird houses or sanctuaries will then correspond to what we call *liking patterns*.

Notably many of these theorems are valid for infinite sets as well. This will be a stepping stone to understanding the main theorems in this article regarding

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

This section is intended to be a stepping stone to understand the main theorems of this article, covered in Section 7. In Section 7 we will replace "pigeons/professors" with "programs," "ice cream flavours" with "inputs," and "liking patterns" with "languages;" the main difference is that in Section 7 we will allow three different results ($\texttt{yes}, \texttt{no}, \texttt{loops}$) when a program acts on an input, as opposed to this section where each pigeon/professor declares one of two results ($\texttt{yes}, \texttt{no}$) when asked about whether or not it/they like a certain flavour of ice cream.

5.1. **The Classical Pigeon Hole (or Drawers) Principle and Co-Principle, and Other Variants.** The classical *pigeon hole* principle states that if some $n \in \mathbb{N} = \{1, 2, \ldots\}$ there are $n+1$ pigeons, each one sheltered in one of n bird sanctuaries, then some sanctuary shelters at least two pigeons. Of course, a similar principle holds if each of $n+1$ professors are sheltered in one of n bird santuaries.

There are a number of variants. For example, if each of $2n+1$ professors are sheltered in n bird sanctuaries, then some sanctuary shelters at least three professors.

The varaint that is of most interest to us will be that if each of n professors is sheltered in $n+1$ bird sanctuaries, then some sanctuaries is free of professors. We will call this the *pigeon hole co-principle*.

5.2. **Finite Counting with Professors or Pigeons and Ice Cream.** In this subsection we give some "counting principles" for finite sets to give some intuition for our discussion of counting principles for infinite sets. This includes two versions of Cantor's theorem that are valid for finite and infinite sets.

The reader may keep in mind that:

(1) our notion of "professor" or "pigeon" will later be replaced by that of a "program;" the set of all professors, pigeons, and/or programs will be denoted by $\mathcal{P}$;
(2) similarly for "ice cream flavours" and "inputs (to a program)," the set of all of which is denoted by $\mathcal{I}$;
(3) similarly for "liking patterns" and "languages," the set of all of which is denoted by $\mathcal{L} = \text{Power}(\mathcal{I})$, i.e., the set of all subsets of $\mathcal{I}$.

5.2.1. *An Informal "Pigeon Hole" Principle.* Imagine that we have a set, $\mathcal{I}$, of 5 flavours of ice cream: (α) strawberry, (β) coffee, (γ) broccoli-and-cheddar, (ϵ) coq-au-vin, (ζ) hummus (vegan). We use $\text{Power}(\mathcal{I})$ to denote the set of all subsets of $\mathcal{I}$; $\text{Power}(\mathcal{I})$ therefore has 32 elements, and can be arranged be arranged in a number of ways, e.g. in increasing size of subsets:

$$\begin{aligned}\text{Power}(\mathcal{I}) = \Big\{&\emptyset,\\ &\{\alpha\}, \ldots, \{\zeta\},\\ &\{\alpha,\beta\}, \{\alpha,\gamma\}, \ldots, \{\epsilon,\zeta\},\\ &\\ &\vdots\\ &\\ &\{\alpha,\beta,\gamma,\epsilon,\zeta\}\Big\}\end{aligned}$$

Say that we have a set, $\mathcal{P}$, whose elements are professors, and each $p \in \mathcal{P}$ likes some flavours, and dislikes others; i.e., we have a function $\text{Result}\colon \mathcal{P} \times \mathcal{I} \to$

$\{\texttt{yes}, \texttt{no}\}$, with the following meaning:

$$\begin{aligned} \text{Result}(p,i) = \texttt{yes} \quad &\text{“means”} \quad \text{prof } p \text{ likes ice cream } i, \text{ and,} \\ \text{Result}(p,i) = \texttt{no} \quad &\text{“means”} \quad \text{prof } p \text{ dislikes ice cream } i. \end{aligned}$$

Here are some type of “pigeon hole” principles, where the professors play the role of the pigeons.

Proposition 5.1. *Let* $\mathcal{P}, \mathcal{I}, \text{Result}$ *be as above.* $\text{Result}\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}\}$ *where* $\mathcal{I} = \{\alpha, \beta, \gamma, \epsilon, \zeta\}$ *and* $|\mathcal{P}| \geq 33$*. Then there are two professors,* $p_1, p_2 \in \mathcal{P}$ *that have the same* ice cream liking pattern*, i.e., such that*

$$\forall i \in \mathcal{I}, \quad \text{Result}(p_1, i) = \text{Result}(p_2, i).$$

Notice that the term *ice cream liking pattern* is in italics, because we have not formally defined this notion. Notice that the phrase “be as above” is a bit vague. Let us make this more precise, at the risk of being mildly tedious, and generalize the setting.

5.2.2. *A Formal “Pigeon Hole” Principle.*

Definition 5.2. By a *Professor-Ice Cream system* we mean the data consisting of a sets $\mathcal{P}, \mathcal{I}$ and a map $\text{Result}\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}\}$. (More formally, a Professor-Ice Cream system is a triple $(\mathcal{P}, \mathcal{I}, \text{Result})$ as above.) We refer to $\mathcal{P}$ as the *set of professors* (or, at times, the *set of pigeons*), and the elements of $\mathcal{P}$ as *professors* or *pigeons*. We refer to $\mathcal{I}$ as the *set of ice creams* and its elements as *ice creams*. We refer to $\mathcal{L} = \text{Power}(\mathcal{I})$ as the *set of liking patterns* of the Professor-Ice Cream system, and its elements as *liking patterns*. For any $p \in \mathcal{P}$, we define the *liking pattern of professor/pigeon* p to be

$$\text{LIKING}(p) \stackrel{\text{def}}{=} \{i \in \mathcal{I} \mid \text{Result}(p,i) = \texttt{yes}\}.$$

(In [Sip] and similar textbooks, it is common to use all caps for the names of languages; hence we follow this convention, when reasonably, for liking patterns as well.)

Throughout the discussion below, if $\mathcal{S}$ is a set, then $|\mathcal{S}|$ is the size (i.e., number of elements) of $\mathcal{S}$. (If $\mathcal{S}$ is infinite, then one would usually define $|\mathcal{S}|$ to be the *cardinality* of $\mathcal{S}$; we won't worry about this for now.)

Proposition 5.3. *Let* $(\mathcal{P}, \mathcal{I}, \text{Result})$ *be a Professor-Ice Cream system, where* $\mathcal{I}$ *is a finite set. Then if* $|\mathcal{P}| > 2^{|\mathcal{I}|}$*, then there are two professors with the same likings.*

As we will explain in class, we do not assume that $\mathcal{P}$ is finite; we leave as an exercise to consider the ramifications of having infinitely many professors or pigeons.

5.2.3. *A “Too Few Professors/Pigeons” Principle, i.e., the Professor/Pigeon Co-Principle.* What happens when there are too few professors or pigeons in some part of the world? For example, if you have five flavours of ice cream (as above), but at most 31 professors or pigeons, then (since $31 < 2^5 = 32$) there is some liking pattern that is not found among the 31 professors or pigeons. Let us state this formally.

Proposition 5.4. *Let* $(\mathcal{P}, \mathcal{I}, \text{Result})$ *be a Professor-Ice Cream system, where* $\mathcal{P}$ *is a finite set. Then if* $|\mathcal{P}| < 2^{|\mathcal{I}|}$*, then there is some liking pattern* $L \in \mathcal{L} = \text{Power}(\mathcal{I})$ *that cannot be found among any of the professors (or pigeons).*

5.2.4. *Cantor's Theorem as it Applies to Professors or Pigeons and Ice Cream.* We will now improve Proposition 5.4, in the sense that if $\mathcal{P}, \mathcal{I}$ as above (i.e., in a Professor-Ice Cream system) have the same size, then we can explicitly write down a liking pattern that does not occur. It is essentially the deservedly well-known Cantor's Theorem.

Theorem 5.5. *Let* $(\mathcal{P}, \mathcal{I}, \text{Result})$ *be a Professor-Ice Cream system where* $\mathcal{P} = \mathcal{I}$. *Then the liking pattern*

$$L = \{i \in \mathcal{I} \mid \text{Result}(i, i) = \texttt{no}\} \subset \mathcal{I} \tag{1}$$

is not the liking pattern of any professor/pigeon.

Proof. Assume that $L \subset \mathcal{I}$ is the liking pattern of professor $i' \in \mathcal{I}$ (recall $\mathcal{I} = \mathcal{P}$). Then we have

$$\{i \in \mathcal{I} \mid \text{Result}(i', i) = \texttt{yes}\} = \text{LIKING}(i') = L = \{i \in \mathcal{I} \mid \text{Result}(i, i) = \texttt{no}\}.$$

But this implies that

$$\{i \in \mathcal{I} \mid \text{Result}(i', i) = \texttt{yes}\} = \{i \in \mathcal{I} \mid \text{Result}(i, i) = \texttt{no}\}. \tag{2}$$

Now $\text{Result}(i', i')$ must equal either `yes` or `no`, and we now show that either way (2) gives a contradiction: indeed, if $\text{Result}(i', i') = \texttt{yes}$, then i' lies on the set on the LHS (left-hand-side) of (2) but not on the RHS one, which contradicts (2). Similarly if $\text{Result}(i', i') = \texttt{no}$ we get a similar contradiction. □

Notice that this theorem does not assume that $\mathcal{P} = \mathcal{I}$ is a finite set; furthermore, the proof above works for arbitrary sets.

Example 5.6. Consider three professors, Profs. Strawberry, Coffee, and Hummus, and three flavours of ice cream, strawberry, coffee, and hummus. Prof. Strawberry likes hummus flavoured ice cream (and not the other two flavours), Prof. Coffee likes all three flavours, and Prof. Hummus does not like any of these three flavours. Thinking of this as the case $I = P$, the set L in (1) equals

$$L = \{\text{strawberry},\ \text{hummus}\}$$

Of course, one can easily check that L does not equal any of the three likings of our professors. However, one can give (a much longer proof) of this along the lines of the proof of Theorem 5.5: for example, the liking pattern of Prof. Strawberry is

$$\text{LIKING}(\text{Strawberry}) = \{\text{humus}\},$$

so LIKING(Strawberry) does not contain strawberry ice cream, but L above does. Similarly for the other two profs/pigeons.

Curiously, it may be simpler to state a far more general and far stronger form Theorem 5.5; first we require a definition.

Definition 5.7. If A, B are sets, we write $|A| \leq |B|$ if there exists an injection $g\colon A \to B$, i.e., a map of sets g such that there are no two distinct $a_1, a_2 \in A$ such that $g(a_1) = g(a_2)$.

[For example, a 8-digit student ID number should uniquely identify the student; hence the map from students to 8-digit ID numbers should be an injection. Can you give another example (hopefully a better one...)?]

Theorem 5.8. *Let* $(\mathcal{P},\mathcal{I},\mathrm{Result})$ *be a Professor-Ice Cream system such that there is an injection* $g\colon \mathcal{P}\to\mathcal{I}$*. Then*

(1) there exists a liking pattern, L*, such that*

$$\forall p\in\mathcal{P}\ \textit{(for all } p \textit{ in } \mathcal{P}\textit{)},\quad g(p)\in L \quad\Longleftrightarrow\quad \mathrm{Result}(p,g(p))=\texttt{no} \tag{3}$$

(i.e., for all $p\in\mathcal{P}$*, if* $\mathrm{Result}(p,g(p))=\texttt{yes}$*, then* $g(p)\notin L$*, and if* $\mathrm{Result}(p,g(p))=\texttt{no}$*, then* $g(p)\in L$*); and*

(2) any liking pattern satisfying (3) is not the liking pattern of any prof/pigeon.

The reader should be able to produce a proof of the second part of this theorem based on the proof of Theorem 5.5; curiously, the first part of this theorem is not hard, but the reader will need to specify L explicitly unless they want to invoke the *axiom of choice*.

5.3. **Cantor's Theorem.** The following is the (more) common way of stating Cantor's theorem.

Theorem 5.9. *Let* $\mathcal{I}$ *be a set. Then if* $f\colon\mathcal{I}\to\mathrm{Power}(\mathcal{I})$ *is any map, then the set*

$$L=\{i\in\mathcal{I}\mid i\notin f(i)\}$$

is not in the image of f*. In particular, no map* $f\colon\mathcal{I}\to\mathrm{Power}(\mathcal{I})$ *is a surjection.*

Proof. Assume that L were in the image of f, i.e., that for some $i'\in\mathcal{I}$ we have

$$f(i')=L=\{i\in\mathcal{I}\mid i\notin f(i)\}.$$

Then we have either $i'\in f(i')$ or $i'\notin f(i')$, both of which yield a contradiction (details will be given in class and/or the homework). □

5.4. **Finite Counting with Boolean Formulas of a Given Size.** Let us consider a more serious example. Fix an integer, $n\ge 1$. The *Boolean* n*-cube* is $\mathcal{I}=\{0,1\}^n$; a *Boolean formula of size* k *on* n *variables* is any string on $(,),\wedge,\vee,\neg,x_1,\dots,x_n$ obtained from the following inductive description:

(1) $x_1,\dots,x_n,\neg x_1,\dots,\neg x_n$ are each Boolean formulas of size 1;
(2) a formula of size k is a string of the form if f,g are formulas of sizes k_1,k_2, then $(f\wedge g)$ and $(f\vee g)$ where f,g are formulas of sizes k_1,k_2 with $k_1+k_2=k$.

This is the usual notion of a formula, where the size counts the number of variables (with their repetitions) in the formula. Let $\mathcal{P}_k$ denote the set of Boolean formulas of size at most k. We claim that

$$|\mathcal{P}_k|<k!(4n)^k$$

(this bound is a bit crude). Then there is a Boolean function on n variables (i.e., a function from $\mathcal{I}$ to $\{0,1\}$) that is not computed by any formula of size at most k, provided that

$$|\mathcal{P}_k|\le 2^{|\mathcal{I}|},$$

i.e.,

$$k!(4n)^k\le 2^{2^n}.$$

We easily get

Theorem 5.10. *If* n *is sufficiently large, then there is a Boolean function on* n *variables not computed by any formula of size at most* $2^n/(c\log n)$ *where* c *is a constant independent of* n*.*

5.5. **Infinite Counting.** Consider trying to generalize the discussion of the previous subsection to the case where $\mathcal{P}$ and $\mathcal{I}$ can be infinite. It turns out that Theorems 5.9 and 5.5 hold as is. What is more subtle is how does one make sense of statments like $|A| < |B|$, $|A| = |B|$, etc. ?

Many readers will be aware of some subtleties: for example, if $\mathbb{N} = \{1, 2, 3, \ldots\}$ denotes the natural numbers ([Sip] uses $\mathcal{N}$ for $\mathbb{N}$), and $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ ([Sip] uses $\mathcal{Z}$), then there is an isomorphism

$$f \colon \mathbb{N} \to \mathbb{Z}$$

(given in class and/or the homework). Hence even though $\mathcal{N}$ is a proper subset of $\mathbb{Z}$, they seem to have the same "size."

In fact, we will discuss in class and/or the exercises some of the definitions and facts below. [Most likely we will not cover all of these in this years' CPSC 421/501; the reader may also refer to Appendices A–B for a more leisurely account of the first three topics below.]

(1) a set is said to *countably infinite* if it is isomorphic to $\mathbb{N}$, and any infinite set S that is not countably infinite is said to be *uncountable*; examples of uncountable sets are Power($\mathbb{N}$) by Theorem 5.9.
(2) While infinite sets may seem a bit weird at first, they do follow some very reasonable rules. For example, if A, B are infinite sets that are isomorphic, then Power(A) is isomorphic to Power(B).
(3) In CPSC 421/501 we will be extremely interested in the fact that for any alphabet (i.e., finite, nonempty set), $\mathcal{A}$, the set $\mathcal{A}^*$ of words/strings in $\mathcal{A}$ is *countably infinite*, and hence the set of languages over $\mathcal{A}$, i.e., the set Power($\mathcal{A}^*$), is uncountably infinite. Hence there are always languages over $\mathcal{A}$ that are not decided/recognized/etc. by any set of algorithms, in the sense of a subset of $\mathcal{B}^*$ for some alphabet $\mathcal{B}$, even if we endow these algorithms with five of your favourite oracles and three sources of randomness (or anything else, provided that each element of $\mathcal{B}^*$ recognizes at most one language in Power($\mathcal{A}^*$).
(4) For the typical kinds of set theories one assumes, there is a notion of "size" of infinite sets, which is a so-called system of *cardinal numbers*; in this theory one uses $\aleph_0$ to denote the size, $|\mathbb{N}|$, of $\mathbb{N}$ and similarly

$$\aleph_1 = |\text{Power}(\mathbb{N})|, \quad \aleph_2 = \Big|\text{Power}\big(\text{Power}(\mathbb{N})\big)\Big|, \quad \text{etc.}$$

[Actually, the story of $\aleph_i$ for $i \geq 1$ is a bit more complicated, but under the most common set theoretic axioms (e.g., ZFC, i.e, ZF + axiom of choice) the above is true.] The fact that $\mathbb{N} \times \mathbb{N}$ is isomorphic (as sets) to $\mathbb{N}$ explains the "multiplication rule"

$$\aleph_0 \times \aleph_0 = \aleph_0.$$

For similar reasons one has $\aleph_0 + 1 = \aleph_0$, $\aleph_0 < \aleph_1$, etc.

The following topics are more advanced, and we won't cover them this year in CPSC 421/501; however, these facts may amuse some readers.

(1) It was a long-standing open problem to determine if one could prove the *continuum hypothesis*, i.e., that there does not exist cardinal number strictly between $\aleph_0$ and $\aleph_1$ (which is the cardinality of the real numbers $\mathbb{R}$, hence

the name "continuum"); this was shown to be *independent* of the usual set theory axioms (ZFC) by Paul Cohen in 1963.

(2) Likely a better approach to set theory is the approach of Grothendieck et al. that works with *universes* ([sga72], Exposé I.0). This approach is essentially a refinement of the view that collections should be viewed as one of two types: sets and classes. The theory of universes is far more satisfying and has a number of advantages: e.g., there is a *denumerable (countable)* universe (of finite strings over a fixed alphabet), and its existence stresses the fact that certain constructions involving finite sets yield again finite sets (but some yield infinite sets).

Definition 5.11. If A, B are sets, we write $|A| < |B|$ and say *the size (cardinality) of A is less than that of B* if there is no surjection $f\colon A \to B$.

The above definition is the most convenient definition for us is, since in this case Cantor's Theorem (Theorem 5.9) asserts that for any set, S, $|S| < |\mathrm{Power}(S)|$. However, in most common types of set theory, Definition 5.11 is equivalent to saying that there is no injection $B \to A$, and this equivalence is not at all evident. Similarly, it is easy to see that

$$\left|2^{\mathbb{N}}\right| \le \left|3^{\mathbb{N}}\right| \le \left|4^{\mathbb{N}}\right| = \left|2^{\mathbb{N}}\right|$$

(where $|A| \le |B|$ means that there exists an injection $A \to B$), and hence (in most common set theories) one has

$$\left|2^{\mathbb{N}}\right| = \left|3^{\mathbb{N}}\right|.$$

Yet it is non-trivial to give an isomorphism $2^{\mathbb{N}} \to 3^{\mathbb{N}}$.

6. Why Work with Programs Abstractly? Why Work with Turing Machines?

At the end of this course you should understand the appeal of a Turing machine, despite the fact that it is a highly unrealistic model of a computer (e.g., it has no "random access" memory): namely, it is quite simple to describe in precise mathematical terms. An added bonus is that the class of Turing-machine polynomial time algorithms includes most polynomial time algorithms in the usual "classical" (non-randomized, non-quantum, etc.) sense.

However, even Turing machines take a while to define and get used to. And yet, it is easy to show that there exist unsolvable problems—specifically the *acceptance problem* and the *halting problem*—as soon as one says a small amount about what a "program" or "algorithm" is supposed to be, and how we should be able to build new "programs" or "algorithms" from old ones. This is what we do in the next section. When we cover Sections 3 and 4 of [Sip], we will see essentially the same principles that we now discuss abstractly.

First, let's describe the abstract framework we need in more practical terms.

Consider the problem of given a program written in C (or C++, Python, Javascript, APL, etc.), say whether or not the program will halt, i.e., whether or not it will stop running. We wish to show that if this problem were *decidable*—a concept that we'll say a little more about later—then we get a contradiction. Let us outline how to do this precisely:

(1) **Specify what is meant by a C program:** define an *alphabet*, *symbols* (or *letters* or *characters*) a *string* (or *word*) over an alphabet, and finally a

language over an alphabet (see Chapter 0 of Sipser). The language of valid C programs is a (context-free) language over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots\}$; this language takes a number of pages to specify (as a context-free grammar). The same is true for most programming languages people use these days. (A Turing machine is, in contrast, simply a collection of five-tuples—much easier to specify from scratch.)

(2) **Specify what is meant by running a C program and what is meant by a C program that halts:**

(3) **Convince yourself that C programs are powerful enough to (1) simulate themselves, and (2) do enough interesting things that you consider them to be a reasonable model of what "can be computed:**

(4) **Define technical distinctions such as decided versus recognizing:** If a program takes as input another program and simulates it, the simulator can (1) halt if the simulated program halts, and (2) not halt if the simulated program doesn't halt. So the language HALTING C PROGRAMS[2] can be "accepted." We are claiming it can't be decided, that is there is algorithm/program that takes a C program and *always halts* while correctly saying whether or not the input represents a C program that halts.

(5) **Take the appropriate vague paradox and get the desired theorem:** See Section 5.1 from Sisper's text. Sipser's text uses "TM" or Turing Machines, instead of C Programs. Can you guess why?

7. Program-Input Systems and Uncomputability

We wish to give a summary of the basic theorems about undecidable problems, but in as general a framework as possible. Hopefully this will help clarify Chapters 3 and 4 of [Sip] when we get there. You should make sure that this framework makes sense in the specific examples of Java programs, C programs, Turing Machine programs, etc.

The point of this section is to give all the necessary definitions to prove that the "acceptance" and "halting" problems are undecidable.

Our approach is to bundle all the necessary definitions into two subsections, each definition followed by a/some theorem(s). The key to this section is to understand the definitions and why most classical notations of algorithms—and a few more—satisfy these axioms.

As explained in the last section, computer scientists like to use all capital letters for languages, which helps distinguish them in notation (and is not meant to sound like some computer science theoretician is screaming at you).

7.1. Definition of a Program-Input System, The Language of a Program, Acceptance, Recognition, Deciding, and Lack of Rejection.

Definition 7.1 (Program-Input Systems, Acceptance, Recognition, Deciding)**.** By a *program-input system* we mean a triple

$$\mathcal{S} = (\mathcal{P}_{\mathcal{S}}, \mathcal{I}_{\mathcal{S}}, \text{Results}_{\mathcal{S}}) = (\mathcal{P}, \mathcal{I}, \text{Results}),$$

[2]Computer scientists seem to enjoy writing a specified language in all capital letters, e.g., SAT, 3-COLOR, CLIQUE

where $\mathcal{P}, \mathcal{I}$ are sets, and Results is a map Results: $\mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}, \texttt{loops}\}$. We refer to $\mathcal{P}, \mathcal{I}$ as, respectively, the sets of *programs* and *inputs*, and to Result as the *result function*. We refer to $\mathcal{L} = \text{Power}(\mathcal{I})$ as the *set of languages* of the program-input system. For any $p \in \mathcal{P}$ and $i \in \mathcal{I}$, we say that p *accepts* i if $\text{Result}(p, i) = \texttt{yes}$; otherwise we say that p *does not accept* i (see the explanation below). For any $p \in \mathcal{P}$, we define the *language of the program,* p (or *language recognized by* p) to be

$$\text{LANGUAGE}(p) \stackrel{\text{def}}{=} \{i \in \mathcal{I} \mid \text{Result}(p, i) = \texttt{yes}\} \in \mathcal{L}.$$

We say that $p \in \mathcal{P}$ is a *decider* if for all $i \in \mathcal{I}$, $\text{Result}(p, i)$ is either `yes` or `no`. We say that $L \in \mathcal{L}$ is *recognizable* if for some $p \in \mathcal{P}$ we have $L = \text{LANGUAGE}(p)$, and, moreover, we say that such an L is *decidable* if there exists at least one such a $p \in \mathcal{P}$ that is a decider.

(Compare some of these definitions with Definitions 3.5 and 3.6 of [Sip].)

Remark 7.2. In the notation $\mathcal{S} = (\mathcal{P}_\mathcal{S}, \mathcal{I}_\mathcal{S}, \text{Results}_\mathcal{S})$ above, we frequently drop the subscript $\mathcal{S}$ when confusion is unlikely to occur. **However, there are situations where we will really need the subscript!** For example, in Section 4.1, [Sip] writes

$$A_{\text{DFA}},\ A_{\text{NFA}},\ A_{\text{TM}}$$

for the acceptance languages in the context of, respectively, DFA's, NFA's, and TM's (Turing machines) that are "standardized" in some reasonable way (see below); by constrast, below we would write these languages as

$$\text{ACCEPTANCE}_{\text{DFA}},\ \text{ACCEPTANCE}_{\text{NFA}},\ \text{ACCEPTANCE}_{\text{TM}}.$$

Here it is crucial to keep track of the system, since all three languages are different (the first two languages are decidable but the last one is not). So in the above definition it would be more precise to write:

$$\text{LANGUAGE}_\mathcal{S}(p) \stackrel{\text{def}}{=} \{i \in \mathcal{I}_\mathcal{S} \mid \text{Result}_\mathcal{S}(p_\mathcal{S}, i_\mathcal{S}) = \texttt{yes}\} \in \mathcal{L}_\mathcal{S},$$

and then say that we often drop the subscript $\mathcal{S}$ when confusion is unlikely to occur. However, putting in the subscript $\mathcal{S}$ everywhere becomes rather tedious and cumbersome, both for the writer and the reader. **In this section the letters/symbols $\mathcal{P}, \mathcal{I}$ and words/strings such as** Result, LANGUAGE **are RESERVED WORDS with a special meaning.** If we wanted to be more flexible, we could write $\texttt{yes}_\mathcal{S}$ to allow the particular choice of our "`yes`" word to depend on $\mathcal{S}$; but since we don't need this flexibility, we don't do this; this is why [Sip] omits the blank symbol from the definition of a Turing machine in Chapter 3, although this gets you into trouble in [Sip], Chapter 4, as we will see below.

Notice that in Chapter 1, machines/algorithms (e.g., a DFA, an NFA, etc.) always returns `yes` or `no` as soon as they have read the input; in this context one says that p *rejects* i if $\text{Result}(p, i) = \texttt{no}$. However, Turing machines, C programs, etc., have the possbility of "looping," denoted `loops`, i.e., calculating forever and never halting, never returning a `yes` or `no` (not necessarily stuck in a single infinte loop). In this context one generally avoids the terminology "reject," since if a machine/algorithm does not accept an input, both `no` and `loops` are possible results.

Hence in this section one distinguishes the set of machines/algorithms that are *deciders*, that always halt after a finite amount of time and give a `yes` or `no` answer; a machine/algorithm that is not a *decider* must therefore loop on at least one input.

Notice that a program-input system does not necessarily have to have anything to do with programs or algorithms. For example, we could have

$$\mathcal{P} = \mathcal{I} = \{\texttt{rock}, \texttt{paper}, \texttt{scissors}\},$$

and Result(x, y) could be `yes` if x beats y (in the game Rock, Paper, Scissors), `no` if y beats x, and `loops` if $x = y$.

Let us briefly summarize the terminology in this section for a program-input system:

(1) in our abstract setting we use the words `yes`, `no`, and `loops`, respectively for the words "accepts," "rejects," and "loops" in [Sip];
(2) a **language** is an element of Power$(\mathcal{I})$, which we tend to write in all caps;
(3) a **decision problem** can be viewed either as (1) a language over some alphabet Σ, or (2) (informally) the problem of "computing" a function $\Sigma^* \to \{\texttt{yes}, \texttt{no}\}$;
(4) each program, $p \in \mathcal{P}$ **recognizes** exactly one language, which we denote LANGUAGE(p);
(5) a program is a **decider** if it always **halts**;
(6) a language is **decidable** if it is recognized by at least one **decider**;
(7) unlike DFA's and NFA's, our abstract setting and Turning machines have the possibility of not halting, which we say gives the result **loops**, which [Sip] calls "looping," which does imply that the algorithm has a sort of "infinite loop" in its code.

7.2. **The Existence of an Unrecognizable Language.** The following theorem requires only a small part of Definition 7.1, and uses only Cantor's theorem and the sizes of $\mathcal{P}, \mathcal{I}$, nothing about the particular Result function we choose.

Theorem 7.3. *Let* $(\mathcal{P}, \mathcal{I}, \text{Result})$ *be a program-input system, with* $\mathcal{P}, \mathcal{I}$ *both countably infinite, or, more generally,* $|\mathcal{P}| \leq |\mathcal{I}|$. *Then some language of this system is not recognizable.*

Proof. According to Theorem 5.8, no map $f \colon \mathcal{P} \to \mathcal{L} = \text{Power}(\mathcal{I})$ can be a surjection. Hence the map taking

$$p \mapsto \text{LANGUAGE}(p)$$

is not a surjection, meaning some element of $\mathcal{L}$ is not in the image of $\mathcal{P}$. □

7.3. **The Acceptance Problem is Undecidable.** Consider a C (C++, Python, etc.) program: likely you represent this as an ASCII string, and hence think of $\mathcal{P}$—the set of all C programs—as a subset of ASCII strings (not all ASCII strings are valid programs...). Consider when an input to such a program is an arbitrary ASCII string. Hence

$$\mathcal{P} \subset \text{ASCII}^* = \mathcal{I}$$

(see Chapter 0 in [Sip] and/or Appendix A here to recall this notation and what is meant by an alphabet, a string, etc.).

Provided that a program-input system is expressive enough, one can easily produce undecidable problems, i.e., languages that are undecidable.

Definition 7.4. By an *expressive program-input system* we mean a five-tuple $\mathcal{S} = (\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ where

(1) $(\mathcal{P}, \mathcal{I}, \text{Result})$ is a program-input system.

(2) EncodeProg: $\mathcal{P} \to \mathcal{I}$ and EncodeBoth: $\mathcal{P} \times \mathcal{I} \to \mathcal{I}$ are both injections such that the following conditions hold:

Program Negation: given any program $p \in \mathcal{P}$, there is a program $p' \in \mathcal{P}$, such that

$$\text{Result}(p', i) = \neg\text{Result}(p, i),$$

where $\neg$, called *the negation operation*, is defined via: $\neg\texttt{yes} = \texttt{no}$, $\neg\texttt{no} = \texttt{yes}$, and $\neg\texttt{loops} = \texttt{loops}$; and

Feeding a Program to Itself: Given any program, $p \in \mathcal{P}$, there is a program, $p' \in \mathcal{P}$, such that

$$\forall q \in \mathcal{P}, \quad \text{Result}(p', \text{EncodeProg}(q)) = \text{Result}(p, \text{EncodeBoth}(q, \text{EncodeProg}(q)))$$

For such a system we define the *Acceptance Problem* to be the language given by

(4) $$\text{ACCEPTANCE} = \{\text{EncodeBoth}(p, i) \mid p \in \mathcal{P},\ i \in \mathcal{I} \text{ and } \text{Result}(p, i) = \texttt{yes}\}.$$

We say that $u \in \mathcal{P}$ is a *universal program* if u recognizes ACCEPTANCE. We say that a $p \in \mathcal{P}$ *halts* on an input $i \in \mathcal{I}$ if $\text{Result}(p, i)$ is either $\texttt{yes}$ or $\texttt{no}$. We define

$$\text{HALTING} = \{\text{EncodeBoth}(p, i) \mid p \in \mathcal{P},\ i \in \mathcal{I}, \text{ and } \text{Result}(p, i) = \texttt{yes} \text{ or } \text{Result}(p, i) = \texttt{no}\}.$$

We remark that in practice it is simple to perform "Program Negation," since all you need to do is swap the answers $\texttt{yes}$ and $\texttt{no}$. However, "Feeding a Program to Itself," is more tedious to do, but not particularly difficult in most settings such as C programs, Python programs, Turning machines, etc. We will say a bit more about this in class.

Theorem 7.5. *In any expressive program-input system, the Acceptance Problem, ACCEPTANCE, is undecidable.*

In 2021, we omitted the following proof. However, for historical reasons, we include this proof; if you'd like to see why this is essentially a diagonalization argument, see pages 208–209 in [Sip] (3rd Edition).
This proof tries to keep the same notation as [Sip]. I recommend skipping this proof, since there is a much more concrete way to state the above theorem once we define universal programs; I find the proof below unnecessarily confusing.

Proof. Assume to the contrary that there is some expressive program-input system, $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$, in which some program, $h \in \mathcal{P}$, decides ACCEPTANCE, i.e., h is a decider and

$$\forall p \in \mathcal{P},\ i \in \mathcal{I}, \quad \text{Result}(h, \text{EncodeBoth}(p, i)) = \text{Result}(p, i).$$

According to our ability to "Feed a Program to Itself," there exists a program $h' \in \mathcal{P}$ such that

$$\forall q \in \mathcal{P}, \quad \text{Result}(h', \text{EncodeProg}(q)) = \text{Result}(h, \text{EncodeBoth}(q, \text{EncodeProg}(q))) = \text{Result}(q, \text{EncodeProg}(q)).$$

By our ability to "Negate a Program," there is $d \in \mathcal{P}$ such that

$$\forall q \in \mathcal{P}, \quad \text{Result}(d, \text{EncodeProg}(q)) = \neg\text{Result}(q, \text{EncodeProg}(q)).$$

In particular, setting $q = d$ we have

$$\text{Result}(d, \text{EncodeProg}(d)) = \neg\text{Result}(d, \text{EncodeProg}(d)).$$

According to our rules regarding negation, we must have

$$\text{Result}(d, \text{EncodeProg}(d)) = \texttt{loops}, \tag{5}$$

since only `loops` equals its own negation. But then d is not a decider. □

Note that by the *Halting Problem* one can mean a number of related problems. For example, when $\mathcal{I}$ is the set of strings/words over an alphabet, there is a distinguished "empty input." One could consider the programs that halt on the empty input and define this as "the Halting Problem." We have described the notion of *halting* and the *Halting Problem* at this point simply to explain some common parlance in computer science. Most textbooks that I know actually begin by proving that the Acceptance Problem is undecidable, and then progress from there to the Halting Problem (which can be proven similarly). At present (Fall 2021) I'm not sure if one needs additional assumptions to show that the Halting Problem is undecidable in this abstract setting of this section.

7.4. **Universal Programs and Delightful Programs.**

Definition 7.6. Let $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ be an expressive program-input system. We say that $u \in \mathcal{P}$ is a *weak universal program* if u recognizes ACCEPTANCE, and a *universal program* if

$$\forall p \in \mathcal{P},\ i \in \mathcal{I}, \quad \text{Result}(u, \text{EncodeBoth}(p, i)) = \text{Result}(p, i).$$

In class will explain that term "strong universal program" is the usual definition used in a *universal Turing machine*, but (see below) in most settings one can build a strong universal program by running two weak ones "in parallel."

In other words, a universal program takes as its input both the description of a program, p, and an input, i, and is responsible to produce the same result as the result of p on i. These can built on top of what one usually calls a "debugger" for a program: to find bugs (errors) in a computer program, it is helpful to have a tool that simulates each "step" of a computer program and allows one to examine the contents of each variable at each step. By running a debugger on a computer program, one can "simulate" the running of a program with various inputs, step-by-step. A universal machine is built by running a debugger on a program and its inputs, stopping when the simulation stops and giving either the answer `yes` or `no`; if the simulation never stops, then program is "looping" on its input, and the simulation never stops (which should be what it means to produce the result `loops`).

One can build—with sufficient will and patience—a debugger for most computers, and hence build a universal program. The textbook [Sip] tends to gloss over this point, but I tend to cover this in more detail in class.

Theorem 7.5 can be stated as a "positive theorem[3]."

First note that if $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ is any expressive program-input system, the two conditions in Definition 7.4 allow us to "build" from any universal progrma, u, a program $d \in \mathcal{P}$ such that

$$\forall q \in \mathcal{P}, \quad \text{Result}(d, \text{EncodeProg}(q)) = \neg\text{Result}(q, \text{EncodeProg}(q)); \tag{6}$$

[3]We thank Yuval Peres who stresses that it is often better to prove a theorem without merely deriving a contradition an assumption, but as a result: e.g., rather than proving that there are infinitely many primes, one can prove that the sums of reciprocals of the prime numbers must diverge.

we say "build" because although the existence of d is guarenteed by the two condition in Definition 7.4, in software one can u is a program, and then d can be built using u as a subroutine, where we first feed an input q to itself, then run u, then negate the answer.

Definition 7.7. If $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ is an expressive program-input system, we say that $d \in \mathcal{P}$ is *delightful* if it satisfies (6).

Theorem 7.8. *For any delightful program, d, in any expressive program-input system, $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$,*

$$\text{Result}(d, \text{EncodeProg}(d)) = \texttt{loops}. \tag{7}$$

Proof. Setting q to d in (6), we have

$$\text{Result}(d, \text{EncodeProg}(d)) = \neg\text{Result}(d, \text{EncodeProg}(d)),$$

which immediately implies (6). □

Versions before Nov. 17, 2021 had the results above stated as one long, awkward theorem.

7.5. **Example: Standardized Alphabets and Turing Machines.** Let us summarize some class discussion.

In class we defined a *standardized Turing machine* to be a Turning machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \texttt{blank})$ (the textbook [Sip] omits the blank symbol) of the following form:

(1) $\Sigma = \{1, \dots, s\}$ for some $s \in \mathbb{N}$ (call such an alphabet a *standardized alphabet*)
(2) $Q = \{1, \dots, q\}$ for some $q \in \mathbb{Z}$, with $q_0 = 1$, $q_{\text{acc}} = 2$, $q_{\text{rej}} = 3$;
(3) $\Gamma = \{1, \dots, \gamma\}$ for some $\gamma \in \mathbb{Z}$ (hence $\Sigma \subset \Gamma$), and where the blank symbol, `blank`, is equal to $s + 1$.

The main points are:

(1) by "renaming" or "numbering," any alphabet has a bijection to a *standardized* alphabet, i.e., one of the form $\{1, \dots, s\}$;
(2) by "renaming" or "numbering," any Turing has a bijection to a *standardized* Turning machine as above, and
(3) if we "rename" or "number" the states of a Turing machines, we get the same "algorithm" in the sense that we get the same result over any input.

If

$$\Sigma_{\text{Wow!}} = \{0, 1, \#, L, R\}, \tag{8}$$

we explained how to express any standardized Turning machine as a string over $\Sigma_{\text{Wow!}}$ (the symbols/letters L, R are only there for readability). Furthermore, we explained how to describe an input to such a Turing machine, using the subalphabet $\{0, 1, \#\}$. This gives us functions $\text{EncodeProg}\colon \mathcal{P} \to \Sigma^*_{\text{Wow!}}$ and $\text{EncodeBoth}\colon \mathcal{P} \to \Sigma^*_{\text{Wow!}}$.

7.6. **Oracle Turning Machines and the Acceptance Hierarchy.** In class we wll explain on Tuesday, November 23 that if $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \texttt{blank})$ is any Turning machine, and $f\colon \Gamma^* \to \Gamma^*$ is any function, then we can endow a Turing machine with the ability to call f by writing a word/string w over Γ on some designated tape (of a multitape TM), and if the Turing machine finds itself in a special state, q_{oracle}, "magically" $f(w)$ appears on this tape in place of w. This is like the magical `sin` button (`cos`, `exp`, etc.) on your scientific calculator that you may have been taking for granted since the late 1970's, or—if you have a sense of perspective—the 1980's or early 1990's.

We call such a Turing machine "a Turing machine, M with oracle f," with Γ understood. In practice, we often insist that f is defined one some fixed alphabet Σ_{oracle}, which must be a subalphabet of the tape alphabet (i.e., of $\Gamma = \Gamma_M$) of any Turing machine M that is permitted to call f. We refer to this expressive program-input setting, $\mathcal{S}$, as *the setting of Turing machines endowed with oracle f*. We use the shorthand

$$\text{ACCEPTANCE}_{\text{TM oracle } f} \quad \text{or} \quad \text{ACCEPTANCE}_{\text{oracle } f}$$

to denote the acceptance problem for standardized oracle Turing machines that call the oracle f, once we have standardized such Turing machines in a reasonable way (see Exercise 8.7.2), and similarly define

$$\text{HALT}_{\text{oracle } f}.$$

In practice, we often insist that f maps Γ^*_{oracle} to some two element subset of Γ^*, say $\{w_1, w_2\}$, whereupon we identify identify f with the subset, A, of Γ^*_{oracle} mapping to, say w_1. In this case the oracle is really solving a *decision problem*. For some reason, the oracle then goes in the superscript (ack!), so if $A \subset \Gamma^*_{\text{oracle}}$ is associated to the function $f\colon \Gamma^*_{\text{oracle}} \to \{w_1, w_2\}$, we use

$$\text{ACCEPTANCE}^A \quad \text{refers to} \quad \text{ACCEPTANCE}_{\text{oracle } f},$$

to denote the acceptance problem for standardized Turning machines with oracle A, which assumes some reasonable standardization, such as in Exercise 8.7.2; similarly

$$\text{HALT}^A \quad \text{refers to} \quad \text{HALT}_{\text{oracle } f},$$

the denotes halting problem for standardized Turning machines with oracle A.

There are some minor additional assumptions, such as we need to identify the alphabet $\Sigma_{\text{Wow!}}$ in (8) with some subset of the oracle alphabet Σ_{oracle} so that we can properly query our "oracle" about the acceptance problem.

If we fix some alphabet Γ^*_{oracle}, we easily see that (once we unwind all the definitions, see Exercise 8.7.4) that there is a hierarchy

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

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

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

where for an oracle, A, $\text{Decidable}(A)$ denotes the acceptance problem for standardized oracle TM's with oracle call to A. Similarly there is a "halting hierarchy" with "ACCEPTANCE" replaced everywhere with "HALT."

7.7. **UNDER CONSTRUCTION: The Complement of the Acceptance Problem is Unrecognizable.** Given two computer programs p, p' in any classical setting, we can run p and p' on the same input, i, simultaneously, say one step at a time of each, and halt as soon as one of p or p' halts. This should make the assumption reasonable.

Definition 7.9. We say that an expressive program-input system $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ *has weak simultaneity* if for all $p, p' \in \mathcal{P}$ there is a program p'' such that for all $i \in \mathcal{I}$,

(1) $\text{Result}(p, i) = \texttt{yes}$ and $\text{Result}(p', i) = \texttt{loops}$ implies that $\text{Result}(p'', i) = \texttt{yes}$; and
(2) $\text{Result}(p, i) = \texttt{loops}$ and $\text{Result}(p', i) = \texttt{yes}$ implies that $\text{Result}(p'', i) = \texttt{no}$.

Notice that in most applications, where, say, $\mathcal{I} = \text{ASCII}^*$, the complement of the Acceptance Problem includes (1) those strings that are not meaningful encodings of both a program and in input to the program, and (2) those strings that are meaning encodings of a program, p, and an input i, such that p does not (halt and) return `yes` on input i.

Theorem 7.10. *Let* $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ *be an expressive program-input system with weak simultaneity, for which a universal program exists. Then the complement of the Acceptance Problem, i.e.,*

$$\text{ACCEPTANCE} - \text{COMP} \stackrel{def}{=} \mathcal{I} \setminus \text{ACCEPTANCE}$$

is unrecognizable.

The proof is an easy argument by contradition.

Definition 7.11. We say that an expressive program-input system $(\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$ is *loopable* if for any $p \in \mathcal{P}$ and any function $g\colon \{\texttt{yes}, \texttt{no}, \texttt{loops}\} \to \{\texttt{yes}, \texttt{no}, \texttt{loops}\}$ for which $g(\texttt{loops}) = \texttt{loops}$, there is a program p' such that for all $i \in \mathcal{I}$ such that for all $i \in \mathcal{I}$ we have

$$\text{Result}(p', i) = g\big(\text{Result}(p, i)\big).$$

Notice that "loopable" is a strengthening of the ability to negate a program.

It is easy to see that any expressive program-input system that is loopable and has ETC.

[ADD CONTENT HERE: REMARKS, ETC.]

7.8. **UNDER CONSTRUCTION: Intuitive Summary of Terminology Regarding Expressive Program-Input Systems.** Here is the summary of our terminology for definitions regarding expressive program-input systems,

$$\mathcal{S} = (\mathcal{P}_{\mathcal{S}}, \mathcal{I}_{\mathcal{S}}, \text{Results}_{\mathcal{S}}, \text{EncodeProg}_{\mathcal{S}}, \text{EncodeBoth}_{\mathcal{S}}),$$

or just

$$= (\mathcal{P}, \mathcal{I}, \text{Result}, \text{EncodeProg}, \text{EncodeBoth})$$

when $\mathcal{S}$ is understood in context. We describe things in intuitive where $\mathcal{P}, \mathcal{I}$ are sets, $\text{Result}\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}, \texttt{loops}\}$ where EncodeBoth etc.

Notice that the abstract definitions don't really require that the programs claimed to exist perform the algorithms in a certain way; the abstract definitions don't distinguish between two programs or algorithms

provided they yield the same results! Hence the summary below refers to the intuitive way we would create such programs if we were writing a program.

(1) $\mathcal{S}$ is said to have **program negation** (Definition 7.4) if we can run any program and negation the result (or find a different algorithm that yields this result).
(2) $\mathcal{S}$ is said to be able to **feed a program to itself** (Definition 7.4) if before running any program we preprocess the input as follows: we check if the input is a desciption of a program, EncodeProg(q), for some $q \in \mathcal{P}$, and if so we (erase this input and) replace this input by EncodeBoth(q, EncodeProg(q)); otherwise, if the input is not of the form EncodeProg(q), the program is allowed to behave as it wishes.
(3) A $u \in \mathcal{P}$ is said to be a **(strongly) universal program** if on an input of the form EncodeBoth(p, i), u gives the result equal to Result(p, i) (we think of this as built the same way that a debugger is built, but we don't distinguish between two programs that produce the same result).
(4) A $u \in \mathcal{P}$ is said to be a **weakly universal program** if recognizes ACCEPTANCE. An example of a weakly universal program is a (strongly) universal program is also a
(5) A $d \in \mathcal{P}$ is said to be **delightul** if the result of d on any input of the form EncodeProg(p) is $\neg$ResultEncodeBoth(p, EncodeProg(p)).
(6) A $d \in \mathcal{P}$ is said to have **weak simultaneity** if
(7) A $d \in \mathcal{P}$ is said to **loopable** (i.e., when it talks if)

7.9. **Further Remarks.** It is interesting to consider how the functions EncodeProg and EncodeBoth should be built, since this has connections to Kolmogorov complexity (as of 2021, aka Chaitin-Kolmogorov, Kolmogorov-Chaitin complexity, SolomonoffKolmogorovChaitin complexity, or descriptive complexity). The textbook [Sip] glosses over the fine points of this issue; I always discuss this when I teach CPSC 421/501, since it helps me visualize what is concretely going on. This point can be discussed with no reference to the above material.

Imagine you wish to communicate two seperate ASCII strings. One way to do this is to agree to use the symbol `$` as a separator, e.g.,

`I love my cache` followed by `I have enough cash`

(we have purposely omitted typical English punctation) becomes

`I love my cache$I have enough cash`

However, with this rule the two sentences:

`I love my $` followed by `I have enough $`

becomes

`I love my $$I have enough $`

which cannot be parsed unambiguously, unless we add some more conventions. For example, we could insist on duplicating each true character, and using `%$` as a separator. Hence

`I love my $` followed by `I have enough $`

becomes

`II lloovve mmyy $$%$II hhaavve eennoouugghh $$`

although this could double the length of the string. What could be more efficient conventions? Some possibilities might appear upside-down in a footnote at some point[4]

8. EXERCISES (FOR THIS YEAR, FALL 2021)

8.1. **Paradox Exercises.**

Exercise 8.1.1. 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."

8.1.1(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

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

$$\texttt{plus times two plus} \quad \text{is meaningless,}$$

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.

8.1.1(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.

8.1.1(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`"?

8.1.1(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.]

---

[4] (1) Agree to having certain "reserved symbols/words," (2) True separator = `$a`, actual $ = `$b`, plus, for extra efficiency, actual $$ = `$c`, etc. (3) ???

8.2. **Pigeon Hole Exercises:** POSSIBLY ADD MORE EXERCISES HERE; PROOFREAD CAREFULLY.

Exercise 8.2.1. The point of this exercise is to generalize the pigeon hole principle and co-principle in fairly simple ways.

8.2.1(a) Assume that a village has exactly 21 profs and 10 bird sanctuaries, and that each village prof resides in a single bird sanctuary. **Briefly** explain—in 75 English words or fewer, and ideally closer to 21—why there must be at least three profs residing in some bird sanctuary.

8.2.1(b) Let $n \in \mathbb{N} = \{1, 2, \ldots\}$, i.e., n is a positive integer. Do the same as in part (a) with 21 and 10 replaced by, respectively, $2n + 1$ and n.

8.2.1(c) Assume that a village has exactly 21 profs and 22 bird sanctuaries, and that each village prof resides in a single bird sanctuary. **Briefly** explain—in 75 English words or fewer, and ideally closer to 20—why there must be at least one empty bird sanctuary.

8.2.1(d) Assume that a village has exactly 20 profs and 22 bird sanctuaries, and that each village prof resides in a single bird sanctuary. **Briefly** explain—in 75 English words or fewer, and ideally closer to 20—why there must be at least two empty bird sanctuaries.

8.2.1(e) Let $k, n \in \mathbb{N} = \{1, 2, \ldots\}$, i.e., n, k are positive integers. Assume that a village has exactly n profs and $n+k$ bird sanctuaries, and that each village prof resides in a single bird sanctuary. **Briefly** explain why there must be at least k empty bird sancuatries.

Exercise 8.2.2. The point of this exercise is to give a gentle introduction to the theory of *block designs*, starting with the pigeon hole principle—a very degenerate special case—and building from there. [As of Sept 2021, Wikipedia uses the notation: X, v, b, r, k, λ for, respectively, the point set, $v = |X|$, the number of blocks, r the number of blocks containing a given point, k the number of points in a block, $\lambda =$ number of blocks containing any 2 (or more generally t) distinct points. Furthermore, we show that if we do not insist on distinct points in blocks, etc., and count *with multiplicities*, then we get some easy but crude inequalities.]

OLDER STUFF: Let $m, n, k \in \mathbb{N}$ be positive. Assume that for some $m \in \mathbb{Z}$, a village has exactly m profs, and n bird sanctuaries. Assume that in September, each prof makes exactly k one-hour visits to the bird sanctuaries of the village (and visits cannot overlap). At first, we assume that profs can visit the same bird sancuatry any number of times.

8.2.2(a) Consider the case $k = 1$. ETC.

8.2.2(b) Let $t \in \mathbb{N}$, and assume that each bird santuary is visited at least t different profs. For which quadruples $m, n, k, t \in \mathbb{N}$ is this possible.

8.2.2(c) Now, we let the proofs visit the same bird sanctuary as many times as they like.

8.2.2(d) Similarly, but count each prof visit "with multiplicites" ADD DEFINITON. ETC.

8.2.2(e) Which combination of the above assumptions makes the problem easiest to solve and why?

Exercise 8.2.3. Put another exercise here.

Exercise 8.2.4. Put another exercise here.

8.3. **Negative Self-Referencing and Related Exercises (new as of Fall 2021).**

Exercise 8.3.1. 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

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

i.e., S is the subset of R that consists of each person who does not love themself[5].

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

8.3.1(b) Explain why if David does not love themself, then the set S cannot equal the set of people whom David loves, i.e., the empty set, regardless of whom anyone else loves.

Exercise 8.3.2. Same as Exercise 8.3.1, 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 8.3.3. Same as Exercise 8.3.1, with the modification that no one loves anyone.

Exercise 8.3.4. Consider the setting in Exercise 8.3.1, with the modification that everyone loves everyone.

8.3.4(a) What is S?

8.3.4(b) Explain why if David loves themself, then the set S cannot equal the set of people whom David loves, i.e., all of R, regardless of whom anyone else loves.

Exercise 8.3.5. Consider the setting of Exercise 8.3.1. Draw a table of who loves whom, and explain why the set S is said to be constructed "by diagonalization."

Exercise 8.3.6. Consider the setting of Exercise 8.3.2. Draw a table of who loves whom, and explain why the set S is said to be constructed "by diagonalization."

Exercise 8.3.7. Consider the setting of Exercise 8.3.3. Draw a table of who loves whom, and explain why the set S is said to be constructed "by diagonalization."

Exercise 8.3.8. Consider the setting of Exercise 8.3.4. Draw a table of who loves whom, and explain why the set S is said to be constructed "by diagonalization."

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

Exercise 8.3.9. 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.

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

8.3.9(a) What is S?
8.3.9(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.
8.3.9(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 8.3.10. Consider the same situation as Exercise 8.3.9. 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

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

8.3.10(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.
8.3.10(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.
8.3.10(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.
8.3.10(d) If $f\colon V \to V$ were any other function—not necessarily a bijection—would part (c) still be true?

Exercise 8.3.11. Explain why the following questions can't be answered either yes (true) or no (false).

8.3.11(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?

8.3.11(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?

8.3.11(c) In a certain village, Breyer judges each person who does not judge themself (and no one else). Does Breyer judge themself?

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

8.4. **Problems Phrased in Terms of Pigeons/Profs and Ice Cream.**

In Fall 2021, one student asked if the diagonal is essential in Cantor's Theorem. The next few exercises address this. See also Theorem 5.8.

These exercises use the terms *injection, bijection, surjection*, and, at times, *(one-to-one) correspondence* (a synonym for bijection); these terms are defined just below Definition 4.2 (page 203) of [Sip].

Exercise 8.4.1. Let $\mathcal{P}, \mathcal{I}$ be sets, and let $\iota\colon \mathcal{P} \to \mathcal{I}$ be an injection. Let Result be a function $\mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}\}$.

8.4.1(a) **Briefly** explain why if ι is an injection of finite sets, there exists at least one $L \subset \mathcal{I}$ satisfying

$$\forall p \in \mathcal{P} \text{ (for all } p \text{ in } \mathcal{P}), \quad \iota(p) \in L \quad \Longleftrightarrow \quad \text{Result}\big(p, \iota(p)\big) = \texttt{no}. \tag{9}$$

8.4.1(b) **Briefly** explain why if ι is a bijection, then the inverse function, ι^{-1}, of ι exists, and (9) is equivalent to

$$L = \{i \in \mathcal{I} \mid \text{Result}\big(\iota^{-1}(i), i\big) = \texttt{no}\}. \tag{10}$$

8.4.1(c) If $\iota\colon \mathcal{P} \to \mathcal{I}$ is an injection, can ι^{-1} exist if ι is not a bijection? **Briefly** explain your answer.

8.4.1(d) If $\mathcal{P}, \mathcal{I}$ are finite sets, how many subsets $L \subset \mathcal{I}$ satisfy (9)? Give a formula involving $|\mathcal{P}|, |\mathcal{I}|$, and briefly justify your formula.

8.4.1(e) Let $L \subset \mathcal{I}$ be any subset of $\mathcal{I}$ satisfying (9). For any $p \in \mathcal{P}$, let LIKING(p) be the subset of $\mathcal{I}$ given by

$$\text{LIKING}(p) \stackrel{\text{def}}{=} \{i \in \mathcal{I} \mid \text{Result}(p, i) = \texttt{yes}\}. \tag{11}$$

Show that for all $p \in \mathcal{P}$, $L \neq \text{LIKING}(p)$.

8.4.1(f) Explain why this means that the map LIKING$\colon \mathcal{P} \to \text{POWER}(\mathcal{I})$ is not surjective.

Exercise 8.4.2. A village has three residents: Prof. Hummus, Prof. Pita, Prof. Falafel, and three foods: hummus, pita, falafel. It is given that:

Prof. Hummus: likes hummus and pita, but not falafel.
Prof. Pita: likes hummus and falafel, but not pita.
Prof. Falafel: likes all three foods.

Let

$$\mathcal{P} = \{\text{Prof. Hummus},\ \text{Prof. Pita},\ \text{Prof. Falafel}\},$$
$$\mathcal{I} = \{\text{hummus},\ \text{pita},\ \text{falafel}\}.$$

8.4.2(a) Let $\iota\colon \mathcal{P} \to \mathcal{I}$ be the map given by

$$\iota(\text{Prof. Hummus}) = \text{hummus}, \quad \iota(\text{Prof. Pita}) = \text{pita}, \quad \iota(\text{Prof. Falafel}) = \text{falafel}.$$

Define Result$\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}\}$ be the function given by

$$\text{Result}(p, i) = \begin{cases} \texttt{yes} & \text{if Prof. } p \text{ likes food } i, \text{ and} \\ \texttt{no} & \text{if Prof. } p \text{ does not like food } i. \end{cases}$$

Represent the Result function as a table, similar to class on Thursday (September 23).

8.4.2(b) Use Exercise 8.4.1 to construct a set $L \in \text{POWER}(\mathcal{I})$ that is not in the image of the function LIKING given by (11).

8.4.2(c) Explain why if $\iota\colon \mathcal{P} \to \mathcal{I}$ is any function such that $\iota(\text{Prof. Hummus}) = \text{hummus}$, then any L satisfying (9) cannot equal LIKING(Prof. Hummus), regardless of the rest of the values of ι.

8.4.2(d) Do the same where Prof. Hummus is replaced by Prof. Falafel, and hummus is replaced by falafel.

Exercise 8.4.3. A village has three residents: Prof. Hummus, Prof. Pita, Prof. Falafel, and four foods: hummus, pita, falafel, and chalva. It is given that:

Prof. Hummus: likes hummus, pita, and chalva, but not falafel.
Prof. Pita: likes hummus, falafel, and chalva, but not pita.
Prof. Falafel: likes all four foods.

[Therefore everyone in the village likes hummus and chalva.] Let

$$\mathcal{P} = \{\text{Prof. Hummus}, \ \text{Prof. Pita}, \ \text{Prof. Falafel}\},$$
$$\mathcal{I} = \{\text{hummus}, \ \text{pita}, \ \text{falafel}, \ \text{chalva}\}.$$

8.4.3(a) Let $\iota\colon \mathcal{P} \to \mathcal{I}$ be the map given by

$$\iota(\text{Prof. Hummus}) = \text{chalva}, \quad \iota(\text{Prof. Pita}) = \text{hummus}, \quad \iota(\text{Prof. Falafel}) = \text{falafel}.$$

Define Result$\colon \mathcal{P} \times \mathcal{I} \to \{\texttt{yes}, \texttt{no}\}$ be the function given by

$$\text{Result}(p, i) = \begin{cases} \texttt{yes} & \text{if Prof. } p \text{ likes food } i, \text{ and} \\ \texttt{no} & \text{if Prof. } p \text{ does not like food } i. \end{cases}$$

Represent the Result function as a table, similar to class on Thursday (September 23).

8.4.3(b) Use Exercise 8.4.1 to construct a set $L \in \text{POWER}(\mathcal{I})$ that is not in the image of the function LIKING given by (11).

8.4.3(c) Explain why if $\iota\colon \mathcal{P} \to \mathcal{I}$ is any function such that $\iota(\text{Prof. Hummus}) = \text{chalva}$, then any L satisfying (9) cannot equal LIKING(Prof. Hummus), regardless of the rest of the values of ι.

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

Exercise 8.5.1. Let $\Sigma = \{1, 2\}$, let $L = \Sigma^*$.

8.5.1(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).

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

8.5.1(c) Write the above standardized Turing machine as a word/string over $\Sigma_{\text{Wow!}}$ as described in class notes (November 16 and 18).

8.5.1(d) Write the above standardized Turing machine as a word/string over $\Sigma_{\text{Wow!}}$ and append to it the input 2121, as described in class notes (November 16 and 18).

8.5.1(e) Explain—without actually writing down the word/string—how to Write the above standardized Turing machine as a word/string over $\Sigma_{\text{Wow!}}$ and append to it the input 212121, as described in class notes (November 16 and 18).

Exercise 8.5.2. Same problem as Exericse 8.5.1 for the language $L = \emptyset$.

Exercise 8.5.3. Same problem as Exericse 8.5.1 for the language L described by the regular expression $1(1 \cup 2)^*$.

Exercise 8.5.4. Same problem as Exericse 8.5.1 for the language L described by the regular expression $(1 \cup 2)^*2$.

Exercise 8.5.5. Is the set of standardized Turing machines countable or uncountable? Explain.

Exercise 8.5.6. 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

8.6. **Exercises on Expressive Program-Input Systems, Universal Programs and Delightful Programs.**

Exercise 8.6.1. Let $\Sigma = \{1, 2\}$, let L be the language described by the regular expression $1(1 \cup 2)^*$.

8.6.1(a) Give a Turing machine recoginizing L.

8.6.1(b) Give a Turing machine recoginizing the complement of L.

Exercise 8.6.2. Let $\Sigma = \{1, 2\}$.

8.6.2(a) Give a Turing machine, M, with the following behaviour:
- (a) on words described by the regular expression $1(1 \cup 2)^*$, the Turing machine halts in the accept state;
- (b) on words described by the regular expression $211(1 \cup 2)^*$, the Turing machine halts in the reject state;
- (c) on words described by the regular expression $222(1 \cup 2)^*$, the Turing machine "loops;" and
- (d) on all other words, your machine can behave as you choose.

8.6.2(b) For the Turing machine, M, you gave, give a machine M' whose result is the negation of the result of M (recall, by definition, $\neg$`yes` = `no`, $\neg$`no` = `yes`, $\neg$`loops` = `loops`, i.e., the negation of "accepts" is "rejects" and vice versa, and the negation of "loops" is "loops").

Exercise 8.6.3. Consider the proof in [Sip] that the acceptance problem is undecidable, page 207. Does the description of the machine H determine the result of H on all inputs, or just some particular inputs? Explain.

INSERT MORE EXERCISES HERE

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

Exercise 8.7.1. Let $\Sigma_{\text{Wow!}}$ be as in (8). Let $\pi\colon \Sigma_{\text{Wow!}} \to [5] = \{1, \ldots, 5\}$ be an arbitrary bijection.

8.7.1(a) If $w = \sigma_1 \ldots \sigma_n \in \Sigma^*_{\text{Wow!}}$ is a word, let

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

Does this give a bijection between elements of $\Sigma^*_{\text{Wow!}}$ and elements of $[5]^*$? Explain.

8.7.1(b) If L is a language over $\Sigma_{\text{Wow!}}$, let

$$\pi(L) = \{\pi(w) \mid w \in L\}. \tag{12}$$

Does this give a bijection between laguages over $\Sigma^*_{\text{Wow!}}$ and languages over $[5]^*$? Explain.

Exercise 8.7.2. Let $s \in \mathbb{N}$, and let $\Sigma = [s] = \{1, \ldots, s\}$.

8.7.2(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`).

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

8.7.2(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.

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

Exercise 8.7.3. Let $A \subset \Sigma^*$ be any fixed language, A, over an alphabet Σ of the form $\{1, \ldots, s\}$ for some $s \in \mathbb{N}$. Let $\mathcal{P}$ be the set of all standardized oracle Turing machines that can make an oracle query to A, standardized appropriately (one way of standardizing is given in the above exercises). Let $\mathcal{I} = \Sigma^*$.

8.7.3(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.

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

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

8.7.3(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 8.7.4. Let $A \subset \Sigma^*$ be any fixed language, A, over an alphabet Σ of the form $\{1, \ldots, s\}$ for some $s \in \mathbb{N}$. Let us further assume that $s \geq 5$, so that we may identify $\Sigma_{\text{Wow!}}$ with a subset of $\Sigma = [s]$, and that we have a standardization of all multitape Turing machines as described in the problems above. Let

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

8.7.4(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.

8.7.4(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).

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

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

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

8.7.4(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 8.7.5. Same exercise as above, except with ACCEPTANCE replaced everywhere with HALT.

Exercise 8.7.6. 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?

8.8. **Insert Additional Exercises Here.** INSERT EXERCISES HERE

9. EXERCISES FROM PREVIOUS YEARS (USING DIFFERENT NOTATION, ETC.)

Here are some homeworks for previous years.

N.B. The terminology and documents have changed over the years. Hence I am including this material to show you how these problems have evolved over the years.

9.1. **Homework 1, 2019.** Please note:

(1) You must justify all answers; no credit is given for a correct answer without justification.
(2) Proofs should be written out formally.
(3) Homework that is difficult to read may not be graded.
(4) You may work together on homework, **but you must write up your own solutions individually.** You must acknowledge with whom you worked. You must also acknowledge any sources you have used beyond the textbook and two articles on the class website.

In these exercises, "the handout" refers to the article "Self-referencing, Uncountability, and Uncomputability" on the 421/501 homepage.

(1) Consider the statement: "Alex cuts the hair of those (and only those) who do not cut their own hair." Is there a problem with this statement? Explain. To which "paradox" in the handout is this similar? Explain.

(2) 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

$$\begin{aligned} \texttt{one plus two times two} \quad &\text{means} \quad 1 + 2 \times 2 = 5, \\ \texttt{two times two times two} \quad &\text{means} \quad 2 \times 2 \times 2 = 8, \end{aligned}$$

and

$$\begin{aligned} \texttt{plus times two} \quad &\text{is meaningless,} \\ \texttt{one plus two times} \quad &\text{is meaningless,} \\ \texttt{one two} \quad &\text{is meaningless;} \end{aligned}$$

each sentence either "means" some positive integer or is "meaningless."

(a) Give two different sentences that both "mean" 10.
(b) Explain why every positive integer is the "meaning" of some sentence with words from W.
(c) Explain why every positive integer n is the "meaning" of some sentence of size at most $C(1 + \log_2 n)^2$ for some constant $C \in \mathbb{R}$ independent of n; your explanation should give a value for C.

(3) Consider the five element set

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

where W is the set in Exercise 2 with the ascribed meanings there, and where `moo` has the following meaning:

(a) if `moo` appears anywhere after the first word of a sentence, then the sentence is meaningless,
(b) if `moo` 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."

(a) What is the meaning of "`moo one`"?
(b) What is paradoxical in trying to ascribe a meaning to "`moo two`"?
(c) To which "paradox" in the handout is this similar? Explain.

(4) Which of the following maps are injections (i.e., one-to-one), and which are surjections (i.e., onto)? Briefly justify your answer.
   (a) $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(x) = x + 1$.
   (b) $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(x) = x^2$.
   (c) $f\colon \mathbb{Z} \to \mathbb{Z}$ given by $f(x) = x + 1$.
   (d) $f\colon \mathbb{Z} \to \mathbb{Z}$ given by $f(x) = x^2$.

(5) 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.

(6) 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].)
   (a) Show that $\mathbb{N}^2$ is countable.
   (b) Show that $\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is countable.

(7) Let $S = \{a, b, c\}$ and let $f\colon S \to \text{Power}(S)$ any function such that
$$a \notin f(a), \quad b \notin f(b), \quad c \notin f(c).$$
   (a) Explain why $f(a)$ cannot be all of S.
   (b) Explain why none of $f(a), f(b), f(c)$ equal S.
   (c) What is the set
$$T = \{s \in S \mid s \notin f(s)\}?$$

(8) Let $S = \{a, b, c\}$ and let $f\colon S \to \text{Power}(S)$ any function such that
$$a \notin f(a), \quad b \in f(b), \quad c \notin f(c).$$
   (a) Explain why $f(b)$ cannot equal $\{a, c\}$.
   (b) Explain why none of $f(a), f(c)$ equal $\{a, c\}$.
   (c) What is the set
$$T = \{s \in S \mid s \notin f(s)\}?$$

(End of Homework Problems to be Submitted for Credit.)

Exercises Beyond the Homework (not for credit, solutions will not be released):

Cancellation Property: (1) We say that a map (of sets) $f\colon S \to T$ has the *left cancellation property* if for any two maps g, h from a set $U \to S$ we have $fg = fh$ (i.e., the map $f \circ g\colon U \to T$ equals the map $f \circ h$) implies that $g = h$. Show that this property holds of f iff f is injective.

(2) Formulate a similar *right cancellation property* for a map $f\colon S \to T$ and show that it is equivalent to f being surjective.

[This exercise shows that the notions of "injective" and "surjective" can be defined just in terms of sets and maps (also respectively called *objects* and *morphisms* (or *arrows*) in category theory).]

Unique Positive Rationals: Say that we list the positive rationals—allowing for repitition—as we did in class:

$$1/1, \quad 2/1,\ 1/2, \quad 3/1,\ 2/2,\ 1/3, \quad \ldots$$

Show that as $N \to \infty$, the number of distinct rational numbers in the first N terms of this sequence is

$$N\Big((1-1/4)\ (1-1/9)\ (1-1/25)\ (1-1/49)\ \ldots\Big) + o(N)$$

(i.e., $6N/\pi^2 + o(N)$, using a well-known value of the Riemann Zeta function).

[See the last page of this document for some hints; it is easier to see roughly why the above result is true than to give a rigorous proof of this result.]

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] are 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 $\mathbb{N}$.

Hints for Exercises Beyond: Hints appear on the page after this.

[Hint: It is easier to see why there should be roughly $6N/\pi^2$ distinct rationals in the first N terms than to give a rigorous proof of this result. For a rigorous proof, you could start by showing that the number of a/b in a sequence of length N where a, b are both divisible by two is (1) at most $N/4$ (with no $o(N)$ term) and (2) at least $N/4 + o(N)$. Then consider a, b which are either both divisible by two and/or both divisible by three. Etc. To give a rigorous proof of the $6N/\pi^2 + o(N)$ result you might use the fact that sum of $1/p^2$ over all prime numbers, p, converges, and hence the "infinite tails" of this sum tend to 0.]

(1) Show that a countable union of countable sets is countable; i.e., if $C_1, C_2, \ldots$ are countable sets, show that $C_1 \cup C_2 \cup \cdots$ is countable. If Σ is a countable alphabet, is Σ^* countable?
(2) Let R be the real numbers, and let F be the set of all functions from the reals to the reals. Show that $|R| < |F|$. Can you generalize this statement?
(3) Pick a standard programming language (C, Java, awk, perl, etc.), and outline in a few paragraphs that they satisfy Axioms 1–5. (Axioms 2 and 4 are the only serious issues.)
(4) Does Axiom 5 cover all reasonable functions, f, used in combinations, or does it miss a few? [Hint: What happens if we simply "ignore" Q?] Describe more functions, f, for which Axiom 5 would hold in the example of a standard programming language or Turing machine.
(5) Consider Paradox 4 of Section 1. [This is the basis for Gödel's Incompleteness Theorems.] Argue informally that if (1) any statement in your setting that is provable is true, and (2) no statement in your setting can be both true and false, and (3) you can construct the statement of Paradox 4 in your setting, then this statement must be true but not provably true.
(6) Consider Paradox 3 of Section 1. [This is usually called the "Berry Paradox;" feel free to look it up somewhere.]
   (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

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

$$\texttt{plus times two plus} \quad \text{is meaningless,}$$

   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.
   (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.
   (c) Consider the five element set

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

   with the following meaning for `moo`:

(i) if it appears anywhere after the first word of a sentence, then the sentence is meaningless,
(ii) if it appears only once and at the beginning of a sentence, then we evaluate the rest of the sentence (as usual), and
(A) 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
(B) 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`"?

(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"? [If this last question seems strange or wrong, make up your own version of this question and answer it.]

(7) With axioms and notation in Section 4, let

$$L_{\text{halt}} = \{\text{EncodeBoth}(P, x) \mid P \in \mathcal{P},\ x \in \mathcal{I}, \text{ and } \text{Result}(P, x) \neq \texttt{loops}\}$$

Assuming we have widened Axiom 5 appropriately, as in Exercise 2, show that L_{halt} cannot be decided. [Hint: Assume that it can be decided, and use this to show that L_{yes} can be decided.]

(8) Consider the languages:
(a) Java programs that produce some output;
(b) Java programs that reach line number 40;
(c) Two Java programs that give identical outputs on every input.

Explain why each of these problems is acceptable but not decidable by Java programs (or Turing machines or C programs or ...). [Hint: See the previous exercise to prove that something is not decidable. Notice that we have not made the notion of a "Java program" precise; use may assume any definition and basic properties of Java programs.] [Note that "Java" can be replaced by "C," "awk," "perl," "Turing machine," etc.]

(9) Axioms 3 and 4 are stated in terms of programs, P and P'; however, they don't have much to do with P and P' when dealing with Java programs. For example, P' in Axiom 3 just runs P and then does some Java "post-processing" to change a `yes` to a `no` and vice versa. Similarly Axiom 4 just "pre-processes" to change EncodeProg(Q) to EncodeBoth(Q, EncodeProg(Q)) and then run P. Unfortunately, our axioms and setup doesn't allow a program to produce output other than "yes" or "no" and doesn't allow for composition, i.e., using the output of a first program as input to a second program (as is done in pre- and post-processing).

Write down a set of axioms that generalizes those of Section 4, where Result: $\mathcal{P} \times \mathcal{I} \to \mathcal{I} \cup \{\texttt{loops}\}$ and I contains the elements `yes` and `no` but does not contain `loops`. Make sure Axioms 3 and 4 are not axioms but result from the computability of certain functions from $\mathcal{I}$ to itself, and some sort of "composition axiom."

(10) Consider the following program:

```
10  A = 3^(3^(41^6)) - 2^7
20  COMMENT      Here is a loop
30  IF ( A MOD 2 == 1 ) THEN  A = A * 3 + 1
40  ELSE A = A / 2
50  IF ( A != 1 ) THEN GOTO 30
60  COMMENT      If we get to here then A is 1
70  PRINT("WE ARE DONE")
```

In this programming language, the words IF, THEN, GOTO, ELSE, COMMENT, and PRINT have their intuitive meaning (hopefully it is more or less clear); "GOTO 30" means go to line number 30; line numbers appear at the beginning of a statement; the four operators

```
+,*,MOD,^
```

have their usual meaning. Finally, this programming language is equipped with "long integers" that can grow to be of any size (via some dynamic memory allocation scheme).

Imagine that you'd like to know if lines 60 and 70 of this program are live or dead. Imagine further that you'd like to know this for all variants of line 10 that set A to some fixed value, and that even allows for more complicated procedures instead of lines 30 to 50. Explain why it might be hard to decide (find a method that always gives an answer in finite time) if the program has dead code. Explain why it is definitely undecidable in the case where lines 30 to 50 are replaced by some very complicated function of A.

(11) For a set S and an integer $n \geq 1$, let

$$n^S = \{f \mid f\colon S \to \{0, 1, \ldots, n-1\}\},$$

be the set of all functions from S to $\{0, 1, \ldots, n-1\}$.

(a) Show that for $n = 2$, this definition of 2^S is "the same" as our original definition in class (where 2^S was defined to be the power set of S, i.e., the set of all subsets of S). In other words, give a natural bijection between the power set of S and the set of all functions from S to $\{0, 1\}$.

(b) Show that if $m < n$ are positive integers, there is a natural injection from m^S to n^S for any set S.

(c) Show that there is a simple bijection between $4^{\mathbb{N}}$ and $2^{\mathbb{N}}$.

(d) Explain why the axioms of set theory should imply that there is a bijection between $2^{\mathbb{N}}$ and $3^{\mathbb{N}}$. Do you think it is difficult to describe such a bijection? Explain.

(12) Consider the map α from $2^{\mathbb{N}}$ to the real interval $[0, 1]$ taking $f\colon \mathbb{N} \to \{0, 1\}$ to

$$\alpha(f) = f(1)/2 + f(2)/4 + \cdots + f(n)/2^n + \cdots .$$

By a *diadic rational* we mean a real number of the form $i/2^j$ for some integers i and j.

(a) Show that if r is a diadic rational strictly between 0 and 1, then there are exactly two elements of $2^{\mathbb{N}}$ that map to r under α, and otherwise there is exactly one element.

(b) Let Z be the subset of $2^{\mathbb{N}}$ consisting of those functions, f, such that for some integer, M, we have $f(m) = 0$ if $m \geq M$. Show that α gives a bijection between $2^{\mathbb{N}} \setminus Z$ and $(0, 1]$.
(c) Similarly show that there is a natural bijection between $3^{\mathbb{N}} \setminus Z'$ and $(0, 1]$, where Z' is the subset of $3^{\mathbb{N}}$ consisting of those f such that for some integer, M, we have $f(m) = 0$ if $m \geq M$.
(d) Describe (more or less) a bijection between Z and Z', and use this bijection to give a bijection between $2^{\mathbb{N}}$ and $3^{\mathbb{N}}$.

(13) Show that if $\beta\colon 2^{\mathbb{N}} \to \mathbb{R}$ is defined by

$$\beta(f) = f(1)/3 + f(2)/9 + \cdots + f(n)/3^n + \cdots,$$

then β is an injection. Construct a sujection from $2^{\mathbb{N}}$ to $\mathbb{R}$. Conclude that there is a bijection between $2^{\mathbb{N}}$ and $\mathbb{R}$. Do you think that such a bijection is easy to describe?

(14) Below we shall give examples of $\mathcal{P}, \mathcal{I}$ and a Result function, as in Axiom 1. In each of the examples below, answer the following questions: (1) Can you find functions EncodeProg and EncodeBoth that make Axiom 2 satisfied? If so, what is $L_{\texttt{yes}}$, and what are examples of a universal program, U, stated in Axiom 2? (2) Do Axioms 3, 4, and 6 hold? (3) Assuming Axioms 2–4 hold, what does the theorem that $L_{\texttt{yes}}$ is undecidable mean in this situation? Is it interesting?
(a) $\mathcal{P}, \mathcal{I}$ are arbitrary sets, and Result always returns `yes` (i.e., for all $p \in \mathcal{P}, i \in \mathcal{I}$ Result$(p, i) =$ `yes`).
(b) $\mathcal{P}, \mathcal{I}$ are arbitrary sets, and Result always returns `loops`.
(c) $\mathcal{I}$ is the set of strings of ASCII characters, $\mathcal{P}$ is the set of legal Java programs (or C or BASIC or Fortran or etc.); make your life simple by assuming that a legal Java program cannot have the ASCII string "HERECOMESTHEINPUT" occur anywhere inside it as a substring (this makes EncodeBoth simple). Let Result(p, i) be (1) `yes` if p on input i writes "CPSC 421 is fanstastic." as its first line of output, (2) `no` if p on input i writes "CPSC 421 is great." as its first line of output, and (3) `loops` otherwise.

Appendix A. Decision Problems, Alphabets, Strings, and Languages

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].

A.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[6], 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*[7] to correctly say "yes" or "no" in a finite amount of time.

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

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

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

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

A.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

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

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

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

(1) "1111111," when $\langle n \rangle = \langle n \rangle_2$ means the "binary representation of n" (a unique string over the alphabet $\{0, 1\}$);
(2) "11201," when $\langle n \rangle = \langle n \rangle_3$ means the "base 3 representation of n" (a unique string over the alphabet $\{0, 1, 2\}$);
(3) "one hundred and twenty-seven," when $\langle n \rangle = \langle n \rangle_{\text{English}}$ 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[8], $\langle n \rangle = \langle n \rangle_{\text{Klingon}}$;
(6) and good old "127," when $\langle n \rangle = \langle n \rangle_{10}$ means the "decimal representation of n."

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

A.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 $\mathcal{A}$.

Appendix B. Counting, Power Sets, and Countability

In CPSC 421, for any model of computation that we study (e.g., finite automata, Turing machines, Python programs), we can easily show that there exist programs that cannot be solved. The reason is that there are "more" problems than algorithms.

[Unfortunately, this does not identify which problem(s) cannot be solved, it merely shows that unsolvable problems exist.]

More precisely, we will use Cantor's theorem to show that the set of languages over an alphabet is uncountable. This technique uses *diagonalization* (see Theorem 4.17 and Corollary 4.18 in [Sip]) in a way that looks similar to Russell's famous paradox.

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

B.1. **Injections, Surjections, Bijections, and the Size of a Set.** Any finite set, S, has a size, which we denote by $|S|$. For example,

$$\big|\{a,b,c\}\big| = 3, \quad \big|\{X,Y,Z,W\}\big| = 4.$$

To say that S is *a smaller set* than T, if both are finite sets, just means that $|S| < |T|$.

When working with infinite sets, the notion of one set being *smaller* than another is much more subtle. To compare the "size" of infinite sets one can use the following notions.

Definition B.1. Let $f\colon S \to T$ be a map of sets. We say that f is

(1) *injective* (or *one-to-one*) if for all $s_1, s_2 \in S$ with $s_1 \neq s_2$ we have $f(s_1) \neq f(s_2)$;
(2) *surjective* (or *onto*) if for all $t \in T$ there is an $s \in S$ with $f(s) = t$;
(3) *bijective* (or *a one-to-one correspondence*) if it is injective and surjective.

You should convince yourself that if S, T are finite sets with $|S| < |T|$, then there is no surjective map $S \to T$. See the exercises for related notions.

Definition B.2. Let S, T be two sets. We say that S *is the same size as* T if there is a bijection $S \to T$ (this is Definition 4.12 on page 203 of [Sip]). We say that S *is smaller than* T if there is no surjective map $f\colon S \to T$.

Example B.3. Chapter 0 of [Sip] uses $\mathcal{N}$ to denote the *natural numbers*

$$\mathbb{N} = \{1, 2, 3, \ldots\}.$$

The natural numbers is strict subset of the integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. However, $\mathbb{N}$ and $\mathbb{Z}$ have the same size, since $f\colon \mathbb{N} \to \mathbb{Z}$ can be give as

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

in other words, $f(1) = 0$ and for $k \in \mathbb{N}$, $f(2k) = k$ and $f(2k+1) = -k$.

B.2. **Countable Sets.**

Definition B.4. We say that an infinite set, S, is *countably infinite* if there is a bijection $\mathbb{N} \to S$, i.e., if S is the same size as $\mathbb{N}$. We say that a set is *countable* if it is finite or countably infinite. We say that a set is *uncountable* if it is not countable.

Example B.5. Of course, $\mathbb{N}$ is a countable; since $\mathbb{Z}$ is of the same size as $\mathbb{N}$ (proven above), $\mathbb{Z}$ is countably infinite. In class we will explain why the following sets are also countable:

(1) the positive rational numbers $\mathcal{Q}$ (Example 4.15 in [Sip]);
(2) the rational numbers $\mathbb{Q}$;
(3) the set of words over an alphabet A,

$$A^* = \bigcup_{i \geq 0} A^i$$

(an *alphabet* is any nonempty, finite set).

Example B.6. It is more difficult to prove that a set is uncountable; here are some examples:

(1) the real numbers $\mathbb{R}$ (see Theorem 4.17 in [Sip]);

(2) the set of *languages* over an alphabet, A, meaning the set of all subsets of A^*. We will prove this below; it is also proven as Corollary 4.18 of [Sip], i.e., page 206, although it is not really a "corollary" of the theorem before it.)

B.3. **Cantor's Theorem.**

Definition B.7. If S is a set, the *power set of* S, denoted $\text{Power}(S)$, is the set of all subsets of S.

For example, if S is a finite set with n elements, then its power set has 2^n ("two to the *power* n") elements.

Theorem B.8 (Cantor's Theorem)**.** *Any set, S, is smaller than its power set; i.e., if $f\colon S \to \text{Power}(S)$ is any function, then f is not surjective. Specifically, the set*

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

is not in the image of f.

Proof. For the sake of contradiction, assume that there is a $t \in S$ such that $f(t) = T$. Then either (1) $t \in T$, or (2) $t \notin T$.

In case (1), i.e., if $t \in T$, then we derive a contradiction: in this case

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

which implies that $t \notin f(t)$; but $f(t) = T$, and so $t \notin T$; but this contradicts the assumption that $t \in T$.

In case (2), i.e., if $t \notin T$, then we similarly derive a contradiction: we have

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

which implies that $t \in f(t)$, and hence $t \in T$, which contradicts the assumption that $t \notin T$. □

In class we will explain why this proof uses *diagonalization*; the proof that $\mathbb{R}$ is uncountable is one way to illustrate this. (See Section 4.2 of [Sip].)

Corollary B.9. *Let A be an alphabet. Then the set of languages over A is uncountable. Hence any map from a countable set to the set of decision problems over A is not surjective.*

Proof. The second statement follows from the first and from the definition of uncountable; so it suffices to prove the first statement.

The set of languages over A equals, by definition, $\text{Power}(A^*)$. Let us assume that $\text{Power}(A^*)$ is countable, and derive a contradiction.

The set A^* is countably infinite, and hence there is a bijection $f\colon A^* \to \mathbb{Z}$. If $\text{Power}(A^*)$ were countable, there would exist a surjection $g\colon \mathbb{Z} \to \text{Power}(A^*)$. Then $g \circ f$ would give a sujection $A^* \to \text{Power}(A^*)$. This is impossible by Cantor's theorem. □

Another proof of the corollary above is to use the fact that if $f\colon A \to B$ is a bijection, then f induces a bijection $\text{Power}(A) \to \text{Power}(B)$

B.4. **Unsolvable Problems Exist.** In [Sip] we will describe many notions of what is meant by an "algorithms" (e.g., as described by Turing machines, finite automata, Python programs, C programs, etc.). In most such notions the set of algorithms is countable; for example, a program in any fixed language (Python, C, etc.) is just a finite string.

Assuming that each such algorithm solves a decision problem (i.e., at most one decision problem) over a fixed language, we get a map from algorithms to decision problems.

It follows that from the above corollary that there exist decision problems, i.e., languages over any fixed alphabet, that cannot be solved by any countable set of algorithms.

References

[sga72] *Théorie des topos et cohomologie étale des schémas. Tome 1: Théorie des topos*, Springer-Verlag, Berlin, 1972, Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J. L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat, Lecture Notes in Mathematics, Vol. 269, also available at http://www.cmls.polytechnique.fr/perso/laszlo/sga4/SGA4-1/sga41.pdf. MR 50 #7130

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