Staff Solutions to In-Class Problems Week 5, Mon.

STAFF NOTE: Infinite Cardinality; The Halting Problem, Ch. 7

Problem 1. (a) Several students felt the proof of Lemma 7.1.7 was worrisome, if not circular. What do you think?

Lemma 7.1.7. Let \( A \) be a set and \( b \notin A \). If \( A \) is infinite, then there is a bijection from \( A \cup \{b\} \) to \( A \).

Proof. Here’s how to define the bijection: since \( A \) is infinite, it certainly has at least one element; call it \( a_0 \). But since \( A \) is infinite, it has at least two elements, and one of them must not be equal to \( a_0 \); call this new element \( a_1 \). But since \( A \) is infinite, it has at least three elements, one of which must not equal \( a_0 \) or \( a_1 \); call this new element \( a_2 \). Continuing in the way, we conclude that there is an infinite sequence \( a_0, a_1, a_2, \ldots, a_n, \ldots \) of different elements of \( A \). Now we can define a bijection \( f : A \cup \{b\} \to A \):

\[
\begin{align*}
  f(b) & := a_0, \\
  f(a_n) & := a_{n+1} \quad \text{for } n \in \mathbb{N}, \\
  f(a) & := a \quad \text{for } a \in A \setminus \{a_0, a_1, \ldots\}.
\end{align*}
\]

Solution. There is no “solution” for this discussion problem, since it depends on what seems bothersome.

One issue that puzzles some students (when they are challenged about it) is why the third clause in the definition of \( f \) is needed since \( f \) is already defined on all the \( a_n \)’s. The answer is that there may be elements left over in \( A \), and to be a bijection, the value of \( f \) on each “left-over” element of \( A \) has to be defined somehow. In fact, if \( A \) is uncountable, there are guaranteed to be such left-over elements.

It may also be bothersome that \( f \) is asserted to be a bijection without spelling out a proof. But the bijection property really does follow directly from the definition of \( f \), so it shouldn’t be much burden for a bothered reader to fill in such a proof.

Another possibly bothersome point is that the proof assumes that if a set is infinite, it must have more than \( n \) elements, for every nonnegative integer \( n \). But really that’s the definition of infinity: a set is finite iff it has \( n \) elements for some nonnegative integer, \( n \), and a set is infinite iff it is not finite.

A possibly worrisome point is how you find an element \( a_{n+1} \in A \) given \( a_0, a_1, \ldots, a_n \). But you don’t have to find a specific one: there must be an element in \( A \setminus \{a_0, a_1, \ldots, a_n\} \)—so just pick any one. Actually, the justification for this step is the set-theoretic Axiom of Choice described in the Notes chapter first-order logic, and some logicians do consider it worrisome.

(b) Use the proof of Lemma 7.1.7 to show that if \( A \) is an infinite set, then \( A \surj \mathbb{N} \), that is, every infinite set is “as big as” the set of nonnegative integers.
Solution. By the proof of Lemma 7.1.7, there is an infinite sequence \( a_0, a_1, a_2, \ldots, a_n, \ldots \) of different elements of \( A \). Then we can define a surjective function \( f : A \to \mathbb{N} \) by defining
\[
f(a) := \begin{cases} n, & \text{if } a = a_n, \\ \text{undefined}, & \text{otherwise}. \end{cases}
\]

—A total surjective function is not required, but if you want one define \( f' : A \to \mathbb{N} \), by
\[
f'(a) := \begin{cases} n, & \text{if } a = a_n, \\ 0, & \text{otherwise}. \end{cases}
\]

Problem 2.
Prove that if there is a surjective function \([\leq 1 \text{ out, } \geq 1 \text{ in}] \) mapping \( f : \mathbb{N} \to S \), then \( S \) is countable.

Hint: A Computer Science proof involves filtering for duplicates.

Solution. Think of \( f(0), f(1), f(2), \ldots \) as a stream and filter for duplicates. Namely, define \( g : \mathbb{N} \to S \) recursively by the rules:
\[
g(0) := f(0) \\
g(n + 1) := f(k) 
\text{ where } k \text{ is minimum such that } f(k) \notin \{g(0), g(1), \ldots, g(n)\}.
\]

If \( S \) is infinite, then \( g \) is a bijection. If \( S \) is finite, then it is countable by definition.

Problem 3.
The rational numbers fill the space between integers, so a first thought is that there must be more of them than the integers, but it’s not true. In this problem you’ll show that there are the same number of positive rationals as positive integers. That is, the positive rationals are countable.

(a) Define a bijection between the set, \( \mathbb{Z}^+ \), of positive integers, and the set, \((\mathbb{Z}^+ \times \mathbb{Z}^+)\), of all pairs of positive integers:

\[
(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), \ldots \\
(2, 1), (2, 2), (2, 3), (2, 4), (2, 5), \ldots \\
(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), \ldots \\
(4, 1), (4, 2), (4, 3), (4, 4), (4, 5), \ldots \\
(5, 1), (5, 2), (5, 3), (5, 4), (5, 5), \ldots \\
\ldots
\]

STAFF NOTE: A hint will probably be needed: Suggest “filtering for duplicates” as on Problem 7.9.

Solution. Line up all the pairs by following successive upper-right to lower-left diagonals along the top row. That is, start with \((1,1)\) which is an initial diagonal of length 1. Then follow with the length 2 diagonal \((1,2), (2,1)\), then the length 3 diagonal \((1,3), (2,2), (3,1)\), then the length 4 diagonal \((1,4), (2,3), (3,2), (4,1)\), . . . . So the line up would be

\[
(1, 1) \quad (1, 2) \quad (2, 1) \quad (1, 3) \quad (2, 2) \quad (3, 1) \quad (2, 3) \quad (3, 2) \quad (4, 1) \ldots \\
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \ldots
\]
It’s interesting that this bijection from \((\mathbb{Z}^+ \times \mathbb{Z}^+)\) to \(\mathbb{Z}^+\) happens to have a simple formula. The pair \((k, m)\) is the \(k\)th element on the diagonal consisting of the \(k + m - 1\) pairs whose sum is \(k + m\). The total number of elements in all the preceding diagonals is

\[
0 + 1 + 2 + \cdots + (k + m - 2) = (k + m - 1)(k + m - 2)/2,
\]
so the pair \((k, m)\) is the \((k + m - 1)(k + m - 2)/2 + k\)th element in the line-up.

(b) Conclude that the set, \(\mathbb{Q}^+\), of all positive rational numbers is countable.

**Solution.** To show the positive rationals are countable, we want to show how to line them up in a list. To do this, start with a list of all pairs of positive integers such as the one from part (a). Then, going from left to right, replace each pair \((m, n)\) by the positive rational \(r := m/n\), skipping pairs where \(r\) has already appeared:

\[
1, 1/2, 2, 1/3, 3, 1/4, 2/3, 3/2, 4, \ldots.
\]
This is now the desired list of the positive rationals.

**STAFF NOTE:** Some students may get stuck looking for a nice formula for the \(n\)th positive rational in the list. Warn them not to look for a formula, just a *procedure* to construct the list.

Another, indirect approach is to find surjective functions between \(\mathbb{Z}^+\) and \(\mathbb{Q}^+\) and back, and then appeal to the Schröder-Bernstein Theorem 7.1.4.

To begin, it’s obvious that

\[
\mathbb{Q}^+ \text{ surj } \mathbb{Z}^+.
\]  
(1)

since the identity function restricted to the positive integers does the job. Namely, \(f : \mathbb{Q}^+ \to \mathbb{Z}^+\) where

\[
f(r) := \begin{cases} 
r & \text{if } r \text{ is an integer}, 
\text{undefined} & \text{otherwise},
\end{cases}
\]
is a surjective function.

It’s also obvious that

\((\mathbb{Z}^+ \times \mathbb{Z}^+) \text{ surj } \mathbb{Q}^+\)

since there is a trivial surjective function \(g : (\mathbb{Z}^+ \times \mathbb{Z}^+) \to \mathbb{Q}^+\), namely,

\[
g(m, n) := m/n.
\]

It follows from part (a) that

\[
\mathbb{Z}^+ \text{ surj } \mathbb{Q}^+.
\]  
(2)

Now (1), (2), and the Schröder-Bernstein Theorem 7.1.4 imply

\[
\mathbb{Z}^+ \text{ bij } \mathbb{Q}^+.
\]

**STAFF NOTE:** If there’s extra time, point out the previous approach to the team, namely that by Schröder-Bernstein, all they need show is that \(\mathbb{Z}^+ \text{ surj } \mathbb{Q}^+\) and \(\mathbb{Q}^+ \text{ surj } \mathbb{Z}^+\). Then suggest they try that.
**Hint:** Use Problem 2.

**Supplemental problem:**

**Problem 4.**
Let’s refer to a programming procedure (written in your favorite programming language —C++, or Java, or Python, …) as a *string procedure* when it is applicable to data of type *string* and only returns values of type *boolean*. When a string procedure, $P$, applied to a *string*, $s$, returns *True*, we’ll say that $P$ recognizes $s$. If $\mathcal{R}$ is the set of strings that $P$ recognizes, we’ll call $P$ a recognizer for $\mathcal{R}$.

(a) Describe how a recognizer would work for the set of strings containing only lowercase Roman letters —a, b, . . . , z— such that each letter occurs twice in a row. For example, aaccaabbzz, is such a string, but abb, 00bb, AAbb, and a are not. (Even better, actually write a recognizer procedure in your favorite programming language).

**Solution.** All the standard programming languages have built-in operations for scanning the characters in a string. So simply write a procedure that checks an input string left to right, verifying that successive pairs of characters in the string are duplicated, lowercase roman characters.

A set of strings is called *recognizable* if there is a recognizer procedure for it. So the program you described above proves that the set of strings with doubled letters from part (a) is recognizable.

When you actually program a procedure, you have to type the program text into a computer system. This means that every procedure is described by some string of typed characters. If a string, $s$, is actually the typed description of some string procedure, let’s refer to that procedure as $P_s$. You can think of $P_s$ as the result of compiling $s$.

In fact, it will be helpful to associate every string, $s$, with a procedure, $P_s$. So if string $s$ is not the typed description of a string procedure, we will define $P_s$ to be some fixed string procedure —say one that always returns *False*; so if $s$ is an ill-formed string, $P_s$ will be a recognizer for the empty set of strings.

The result of this is that we have now defined a total function, $f$, mapping every string, $s$, to the set, $f(s)$, of strings recognized by $P_s$. That is we have a total function,

$$f : \text{string} \rightarrow \text{pow(string)}.$$  \hfill (3)

(b) Explain why range($f$) is the set of all recognizable sets of strings.

**Solution.** Since $f(s)$ is the set of strings recognized by $P_s$, everything in range($f$) is a recognizable set. Conversely, every recognizable set is in range($f$): if $\mathcal{R}$ is a recognizable set, then by definition, there is a procedure that recognizes $\mathcal{R}$. Let $r$ be the typed input from which such a recognizer was compiled, that is, suppose $P_r$ is the recognizer for $\mathcal{R}$. This means $\mathcal{R} = f(r) \in \text{range}(f)$.

This is exactly the set up we need to apply the reasoning behind Russell’s Paradox to define a set that is not in the range of $f$, that is, a set of strings, $\mathcal{N}$, that is not recognizable.

(c) Let

$$\mathcal{N} := \{ s \in \text{string} \mid s \notin f(s) \}.$$  

Prove that $\mathcal{N}$ is not recognizable.

**Hint:** Similar to Russell’s paradox or the proof of Theorem 7.1.10.

---

1 The string, $s$, and the procedure, $P_s$, have to be distinguished to avoid a type error: you can’t apply a string to string. For example, let $s$ be the string that you wrote as your program to answer part (a). Applying $s$ to a string argument, say aabbcdd, should throw a type exception; what you need to do is apply the procedure $P_s$ to aabbcdd. This should result in a returned value *True*, since aabbcdd consists of consecutive pairs of lowercase roman letters.
**STAFF NOTE:** Refer students to the Russell paradox slide or the text for Theorem 7.1.10.

**Solution.** By definition of $\mathcal{N}$,

$$s \in \mathcal{N} \iff s \notin f(s).$$

(4)

for every *string*, $s$.

Now assume to the contrary that $\mathcal{N}$ was recognizable by some string procedure. This procedure must have a string, $w$, that describes it, so we have

$$s \in \mathcal{N} \iff P_w \text{ applied to } s \text{ returns True,}$$

$$\iff s \in f(w)$$

(5)

(by def. of $f$)

for all *strings* $s$.

Combining (4) and (5), we have that for every string, $s$,

$$s \notin f(s) \iff s \in f(w),$$

(6)

for all *strings* $s$.

Now letting $s$ be $w$ in (6), we reach the contradiction

$$w \notin f(w) \iff w \in f(w).$$

This contradiction implies that the assumption that $\mathcal{N}$ was recognizable must be false.

---

(d) Discuss what the conclusion of part (c) implies about the possibility of writing “program analyzers” that take programs as inputs and analyze their behavior.

**Solution.** Let’s call a programming procedure “unconscious” if it does not return *True* when applied to its own textual definition.

Rephrased informally, the conclusion of part (c) says that it is logically impossible to design a *general* program analyzer, which takes as input the text of an arbitrary program, and recognizes when the program is unconscious. This implies that it is impossible to write a program which does the more general analysis of how an arbitrary procedure behaves when applied to arbitrary arguments.

By the way, it *is* feasible to write a general procedure that recognizes the texts of “conscious” procedures, namely texts for procedures that *do* return *True* when applied to their own descriptions. The “consciousness” recognizer simply takes an input text, $s$, and then simulates $P_s$ applied to $s$. In other words, this general procedure just acts like a virtual machine simulator or “interpreter” for the programming language of its input programs. Note that this naive procedure runs forever on all input programs that run forever!

It’s also important to recognize that there’s no hope of getting around this by switching programming languages. For example, by part (c), no C++ program can analyze arbitrary C++ programs, and no Java program can analyze Java programs, but you might wonder if a language like C++, which allows more intimate manipulation of computer memory than Java, might therefore allow a C++ program to analyze general Java programs. But there is no loophole here: since it’s possible to write a Java program that is a simulator/interpreter for C++ programs, if a C++ program could analyze Java programs, so could the Java program that simulated the C++ program, contradicting (c).