Math for CS 2015/2019 solutions to "In-Class Problems Week 5, Wed. (Session 11)"

https://github.com/spamegg1

October 23, 2022

Contents

1 Problem 1

1.1 (a)

Several students felt the proof of Lemma 7.1.7 (below) was worrisome, if not circular in the course textbook. 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 $f : A \cup \{b\} \to A$:

$$\begin{aligned} f(b) &::= a_0, \\ f(a_n) &::= a_{n+1} && \text{for } n \in \mathbb{N}, \\ f(a) &::= a && \text{for } a \in A - \{b, a_0, a_1, \ldots\}. \end{aligned}$$

Proof. There is no "solution" for this discussion problem, since it depends on what seems bothersome.

It may be bothersome that the proof asserts that f is a bijection without spelling out a proof. But the bijection property really does follow directly from 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 the set $A - \{a_0, a_1, \ldots, a_n\}$, 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. □

1.2 (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 in the course textbook.

Proof. 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

$$f'(a) = \begin{cases} n & \text{if } a = a_n \\ 0 & \text{otherwise} \end{cases}$$

□

2 Problem 2

Prove that if there is a surjective function (≤ 1 out, ≥ 1 in mapping) $f : \mathbb{N} \to S$, then S is countable.

Hint: A Computer Science proof involves filtering for duplicates.

Proof. Since f is surjective, for every $s \in S$, the set

$$preimage_s ::= \{n \in \mathbb{N} \mid f(n) = s\}$$

is a nonempty subset of $\mathbb{N}$.

By the Well-Ordering Principle, for every $s \in S$, $preimage_s$ has a smallest element $smallest_s$.

Define the set $preimage(S) \subseteq \mathbb{N}$ by

$$preimage(S) ::= \{smallest_s \mid s \in S\}$$

Therefore the function $g : S \to preimage(S)$ defined by $g(s) ::= smallest_s$ is a bijection.

Since $preimage(S) \subseteq \mathbb{N}$, it is a countable set. Therefore $preimage(S)$ is either finite (in which case S is also finite, therefore countable), or there is a bijection $h : preimage(S) \to \mathbb{N}$ (in which case $h \circ g : S \to \mathbb{N}$ is a bijection, therefore S is countable). □

3 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.

3.1 (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:

$$\begin{array}{lllll} (1,1), & (1,2), & (1,3), & (1,4), & (1,5), \quad \ldots \\ (2,1), & (2,2), & (2,3), & (2,4), & (2,5), \quad \ldots \\ (3,1), & (3,2), & (3,3), & (3,4), & (3,5), \quad \ldots \\ (4,1), & (4,2), & (4,3), & (4,4), & (4,5), \quad \ldots \\ (5,1), & (5,2), & (5,3), & (5,4), & (5,5), \quad \ldots \\ \vdots & & & & \end{array}$$

Proof. 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 counts as an initial diagonal of length 1. Then follow the length 2 second diagonal $(1,2),(2,1)$, then the length 3 third diagonal $(1,3),(2,2),(3,0)$, then the length 4 fourth diagonal $(1,4),(2,3),(3,2),(4,1)$, ... So the line up would be

$$\begin{array}{cccccccccccc} (1,1) & (1,2) & (2,1) & (1,3) & (2,2) & (3,1) & (1,4) & (2,3) & (3,2) & (4,1) & \dots \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \dots \end{array}$$

It's interesting that this bijection from $(\mathbb{Z}^+ \times \mathbb{Z}^+)$ to $\mathbb{Z}^+$ has a simple formula: the pair (k,m) is the kth element on the diagonal consisting of the pairs whose sum is $k+m$.

The first diagonal has pairs whose elements have a sum of 2, and contains 1 pair. There are 0 preceding diagonals before it.

The second diagonal has pairs whose elements have a sum of 3, contains 2 pairs. There are 1 preceding diagonals before it.

And so on... The diagonal of pairs with sum $k+m$ contains $k+m-1$ pairs, and there are $k+m-2$ preceding diagonals before it.

So there are $k+m-2$ diagonals BEFORE the diagonal containing (k,m). Therefore, the total number of pairs in all the preceding diagonals is

$$1+2+\ldots+(k+m-2) = (k+m-2)(k+m-1)/2$$

so the pair (k,m) appears as the $(k+m-2)(k+m-1)/2+k$th element in the lineup. Our bijection can be explicitly written as:

$$f((k,m)) ::= (k+m-2)(k+m-1)/2+k$$

(For example $f((4,1)) = (4+1-2)(4+1-1)/2+4 = 6+4 = 10$ as expected.) □

3.2 (b)

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

Hint: Use Problem 2.

Proof. We only need to find a surjective function $i : \mathbb{N} \to \mathbb{Q}^+$. Then by Problem 2, $\mathbb{Q}^+$ is countable.

There is a bijection $f : \mathbb{N} \to \mathbb{Z}^+$ defined by $f(n) ::= n+1$.

By part (a) there is a bijection $g : \mathbb{Z}^+ \to (\mathbb{Z}^+ \times \mathbb{Z}^+)$.

Now consider $h : (\mathbb{Z}^+ \times \mathbb{Z}^+) \to \mathbb{Q}^+$ defined by $h((k,m)) ::= k/m$. This function is surjective.

Define $i : \mathbb{N} \to \mathbb{Q}^+$ by $i = h \circ g \circ f$. Since f, g are bijections and h is a surjection, i is a surjection. □

4 Problem 4

For this problem we will be using the word "string" in two different ways, they are distinguished by the fonts we use: notice the difference between string and ***STRING***. The first refers to a sequence of characters as usual, the other refers to a data type in a programming language.

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 R is the set of strings that P recognizes, we'll call P a recognizer for R.

4.1 (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).

Proof. 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.

(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 ***aabbccdd***, should throw a type exception; what you need to do is apply the procedure P_s to ***aabbccdd***. This should result in a returned value ***True***, since ***aabbccdd*** consists of consecutive pairs of lowercase roman letters.)

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,

$$f\colon \boldsymbol{STRING} \to \text{pow}(\boldsymbol{STRING})$$

4.2 (b)

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

Proof. Since $f(s)$ is the set of strings recogized by P_s, everything in range(f) is a recogizable set. Conversely, every recogizable set is in range(f): if $\mathcal{R}$ is a recognizable set, then by definition, there is a procedure, P, that recognizes $\mathcal{R}$. So if r is the input program from which P was compiled, then $\mathcal{R} = f(r)$. □

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.

4.3 (c)

Let

$$\mathcal{N} ::= \{s \in \boldsymbol{STRING} \mid s \notin f(s)\}$$

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

Hint: Similar to Russell's paradox or the proof of Theorem 7.1.11 in the course textbook.

Proof. By definition of $\mathcal{N}$,

$$s \in \mathcal{N} \text{ iff } s \notin f(s) \quad (2)$$

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

$$\begin{array}{lll} s \in \mathcal{N} & \text{iff} & P_w \text{ applied to } s \text{ returns } \boldsymbol{True}, \\ & \text{iff} & s \in f(w) \qquad \text{(by def. of } f\text{)(3)} \end{array}$$

for all ***STRING***s s.

Combining (2) and (3), we have that

$$s \notin f(s) \text{ iff } s \in f(w), \quad (4)$$

for all ***STRING***s s.

Now letting s be w in (4), we reach the contradiction

$$w \notin f(w) \text{ iff } w \in f(w).$$

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

4.4 (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.

Proof. Let's call a programming procedure "self-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 (textual definition) of an arbitrary program, and recognizes when the program is self-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 some given arguments.

BTW, it is feasible to write a general procedure that recognizes when an arbitrary input procedure does return a value when appiled to the string that describes it, that is, when the procedure is self-conscious. The general procedure appllied to input s just 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.

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).

It's a different story if we think about the practical possibility of writing programming analyzers. The fact that it's logically impossible to write analyzers for completely general programs does not mean that you can't do a very good job analyzing interesting programs that come up in practice. In fact these "interesting" programs are commonly intended to be analyzable in order to confirm that they do what they're supposed to do.

So it's not clear how much of a hurdle this theoretical limitation implies in practice. What the theory does provide is some perspective on claims about general analysis methods for programs. The theory tells us that people who make such claims either:

are exaggerating the power (if any) of their methods, say to get a grant or make a sale, or

are trying to keep things simple by not going into technical limitations they're aware of, or

perhaps most commonly, are so excited about some useful practical successes of their methods, that they haven't bothered to think about their limitations.

So from now on, if you hear people making claims about completely general program analysis/verification/optimization methods, you'll know they can't be telling the whole story. □