EECS 70 Discrete Mathematics and Probability Theory
Fall 2015 Satish Rao
Discussion 8b

1. Countable Set

A set S is countable if there is a bijection between S and $\mathbb{N}$ (set of natural numbers) or some subset of $\mathbb{N}$. If $T = \{\ldots, -4, -2, 0, 2, 4, \ldots\}$, is T countable? If yes, can you find two different bijections between T and $\mathbb{N}$.

Solution: Yes. Two bijections from $\mathbb{N}$ to T can be

$$f_1(x) = \begin{cases} x, & \text{if } x \text{ is even;} \\ -x-1, & \text{otherwise.} \end{cases}$$

and

$$f_2(x) = \begin{cases} -x, & \text{if } x \text{ is even;} \\ x+1, & \text{otherwise.} \end{cases}$$

Of course, given $f_1(x)$, picking two different natural numbers and switching their function values, *i.e.*, setting $f_2(x_1) = f_1(x_2)$ and $f_2(x_2) = f_1(x_1)$, are also fine.

2. Count it!

For each of the following collections, determine and briefly explain whether it is finite, countably infinite (like the natural numbers), or uncountably infinite (like the reals):

(a) The integers which divide 8.

Solution: Finite. They are $\{-8, -4, -2, -1, 1, 2, 4, 8\}$.

(b) The integers which 8 divides.

Solution: Countably infinite. We know that there exists a bijective function $f : \mathbb{N} \to \mathbb{Z}$. Then function $g(n) = 8f(n)$ is a bijective mapping from $\mathbb{N}$ to integers which 8 divides.

(c) The functions from $\mathbb{N}$ to $\mathbb{N}$.

Solution: Uncountably infinite. We use the Cantor's Diagonalization Proof:
Let $\mathscr{F}$ be the set of all functions from $\mathbb{N}$ to $\mathbb{N}$. We can represent a function $f \in \mathscr{F}$ as an infinite sequence $(f(0), f(1), \cdots)$, where the i-th element is $f(i)$. Suppose towards a contradiction that there is a bijection from $\mathbb{N}$ to $\mathscr{F}$:

$$\begin{aligned} 0 &\longleftrightarrow (f_0(0), f_0(1), f_0(2), f_0(3), \cdots) \\ 1 &\longleftrightarrow (f_1(0), f_1(1), f_1(2), f_1(3), \cdots) \\ 2 &\longleftrightarrow (f_2(0), f_2(1), f_2(2), f_2(3), \cdots) \\ 3 &\longleftrightarrow (f_3(0), f_3(1), f_3(2), f_3(3), \cdots) \\ &\vdots \end{aligned}$$

Consider the function $g : \mathbb{N} \to \mathbb{N}$ where $g(i) = f_i(i) + 1$ for $i \in \mathbb{N}$. We claim that the function g is not in our finite list of functions. Suppose for contradiction that it did, and that it was the n-th function $f_n(\cdot)$ in the list, i.e., $g(\cdot) = f_n(\cdot)$. However, $f_n(\cdot)$ and $g(\cdot)$ differ in the n-th number, i.e. $f_n(n) \neq g(n)$, because by our construction $g(n) = f_n(n) + 1$ (Contradiction!).

3. Computability

We say that a computer program P computes the function $f : S \to \{0,1\}$ if, for every $x \in S$, when you run the program P on the input x, the program eventually finishes and returns the value $f(x)$. We say that a function f is *computable* if there exists some computer program that can compute f. (Programs always must be of finite length.)

Each of the following parts defines a function f. For each part, say whether f is computable or uncomputable.

1. $f : \mathbb{N} \to \{0,1\}$, defined by
$$f(a) = \begin{cases} 1 & \text{if } a \text{ is prime,} \\ 0 & \text{otherwise.} \end{cases}$$

   Solution: Computable. It is trivial to write a program to test if a given input $a \in \mathbb{N}$ is prime (note that we don't care about efficiency here). The program can simply enumerate all natural numbers from 2 to $\lfloor \sqrt{a} \rfloor$ inclusive, and output 0 if any of them divides a. If none of these divides a, then it outputs 1.

2. $f : \mathbb{N} \to \{0,1\}$, defined by
$$f(a) = \begin{cases} 1 & \text{if there exist prime numbers } b,c \text{ such that } a = b + c, \\ 0 & \text{otherwise.} \end{cases}$$

   Solution: Computable. We can write a program which given any $a \in \mathbb{N}$, simply iterates through all pairs of integers $(2, a-2), (3, a-3), \ldots, (a-2, 2)$ to check if any are of the desired form. If a pair $(x, a-x)$ is found such that both x and $a-x$ are prime (which can be tested by the subroutine from part (b)), then it outputs 1. If no such pair $(x, a-x)$ is found, it outputs 0.

3. $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, defined by
$$f(P,I) = \begin{cases} 1 & \text{if program } P \text{ ever divides a number by zero at any point when run on input } I, \\ 0 & \text{otherwise.} \end{cases}$$

   This is a hard question, so we will guide you through the proof:

   Suppose `DivideByZeroChecker` be the program that computes f. I.e. given program `P` and input `I` (represented as bit strings), `DivideByZeroChecker(P, I)` returns "1" if `P` ever divides by 0 when run on input `I`, and returns "0" otherwise.

   Construct another program `Contrarian` as follows:

   `Contrarian(P):`
   1. If `DivByZeroChecker(P,P)` = "yes", then return immediately.
   2. Otherwise, divide 1 by 0, and then return.

   What happens when `Contrarian` is passed its own source code as its input, i.e., when we run `Contrarian(Contrarian)`. Does it divide by zero?

   Solution: Uncomputable.

   There are two cases:

   - If `DivByZeroChecker(Contrarian, Contrarian)` returns "yes", then we can see that `Contrarian(Contrarian)` does nothing and returns immediately on line 1. In this case, `DivByZeroChecker` predicted `Contrarian(Contrarian)` would divide by zero, but it actually doesn't: so `DivByZeroChecker` was wrong.

- If `DivByZeroChecker(Contrarian,Contrarian)` returns "no", then we can see that `Contrarian(Contrarian)` divides by 0 on line 2. In this case, `DivByZeroChecker` predicted it wouldn't divide by zero, but it actually did: so `DivByZeroChecker` was wrong in this case too.

In either case, `DivByZeroChecker`'s prediction was wrong. This contradicts our assumption that `DivByZeroChecker` exists and always gives the correct output. Therefore, our assumption must have been mistaken. So f is uncomputable.

4. Compute this

(a) Can you write a program that gets n (a natural number) as input and finds the shortest formula that computes n? A formula is a valid sequence consisting of decimal digits, the operators $+$, $\times$,^(raising to the power), and parentheses. The length of a formula is simply the number of characters you need to use to type it (i.e. each operator, decimal digit, or paranthesis counts as one character).

Solution: Yes it is possible to write such a program. We already know one way to write a formula for n, which is to just write the number n (with no operators). Let the length of this formula in characters be l. In order to find the shortest formula we simply need to search among formulae that have length at most l. But there are a finite number of such formula and we can write a program that iterates over all of them (e.g. by treating each character as a byte or an 8-bit number, the whole formula becomes a binary integer of length at most $8l$, so we can simply iterate over all binary numbers up to 2^{8l} and for each one check if it is a valid formula). For each formula that we encounter we can compute its value in finite time (since there are no loop/control structures in formula). Therefore we can check whether it computes n, and then among those that do compute n we find the smallest one.

(b) (Optional) Now assume that you want to write a computer program that given the input n (a natural number) finds another computer program (in a specific language, e.g. C or Python) that prints n. The program that is found has to have the minimum length plus execution time amongst all programs that print n, where length is measured by the number of characters in the source code and execution time is measured by a concrete number such as the number of CPU instructions executed. Can this be done?

Solution: Yes. Again it is possible to write such a program. Again given a number n there is one way we know that we can write n, which is to print its digits one by one (each using a print statement for example). So we already know of a program that prints n. Let the length plus running time of this program be l (where $l \simeq c \lg n$ for some constant c, but this does not matter). We only need to check programs that have a length of at most l and a running time of at most l (since otherwise their running time plus length would be bigger than l). One can again similarly to the previous part iterate over all programs of length at most l (by treating each one as a large binary integer and checking each one's validity by e.g. compiling it). One might be tempted to again run each program and check whether its output is n, but the problem is that we do not know how long each program runs for (if we knew we could have solved the halting problem which is impossible). But note that programs that have a running time of more than l can simply be discarded. So for each program, we run it for at most l steps. If it takes more time, we stop executing it and go to the next program, otherwise in at most l steps we see its output and we can check whether it is equal to n or not.

Now among all programs that have length at most l and execute for at most l steps and print n we find the one that has the shortest length plus execution time.

(c) Consider the set of programs (or functions) that take a single natural number n as input and output a natural number in at most $10^6 + 2^n$ steps (i.e. they always terminate after $10^6 + 2^n$ steps). Let this set be L. A member of L is called **thorough** if every natural number m can be produced as its output

(by an appropriate input). Can you write a program that takes a member of L as input and determines whether that member is thorough? The given member of L is guaranteed to be in L, there is no need for your program to verify the membership.

(HINT: If you had such a program, could you somehow use it to solve the halting problem? If so, what would that mean?)

Solution: No, this is not possible. We will prove this by contradiction. So assume that this task was possible and we have a program P that takes as its input a member of L and determines whether it is thorough or not. We will show that by using P we can solve the halting problem which we know is impossible, therefore proving that P cannot exist.

To solve the halting problem, consider an input to the halting problem H. We need to determine whether H halts at some point or not. Let us construct a program G that takes a natural number n as its input and simulates H for at most n steps and then prints the number of steps H ran for in the output. The program G is thorough if and only if H never halts. This is because if H halts, then it stops after some finite number of steps l, and G never outputs any number larger than l. But if H does not halt, then on every input n, the program G simulates H for the whole n steps and outputs n. So G prints its input to the output and therefore is thorough.

Note that the program G is in L, because simulating H for n steps takes time at most a small constant times n (e.g. $10n$) which is always smaller than $10^6 + 2^n$. So now by feeding G into P and checking the ouput we can find out whether H was halting or not. This shows that with the help of P we can solve the halting problem. So it means that P cannot exist.