CS 70 Discrete Mathematics and Probability Theory
Fall 2025 Sabin, Hug

DIS 7B

Countability Intro

Note 11

Countability: Formal notion of different kinds of infinities.

- *Countable*: able to enumerate in a list (possibly finite, possibly infinite)
- *Countably infinite*: able to enumerate in an infinite list; that is, there is a bijection with $\mathbb{N}$.

  To show that there is a bijection, the *Cantor–Bernstein theorem* says that it is sufficient to find two injections, $f : S \to \mathbb{N}$ and $g : \mathbb{N} \to S$. Intuitively, this is because an injection $f : S \to \mathbb{N}$ means $|S| \leq |\mathbb{N}|$, and an injection $g : \mathbb{N} \to S$ means $|\mathbb{N}| \leq |S|$; together, we have $|\mathbb{N}| = |S|$.
- *Uncountably infinite*: unable to be listed out

  Use *Cantor diagonalization* to prove uncountability through contradiction.

Sometimes it can be easier to prove countability/uncountability through bijections with other countable/uncountable sets respectively. Common countable sets include $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{N} \times \mathbb{N}$, finite length bitstrings, etc. Common uncountable sets include $[0,1]$, $\mathbb{R}$, infinite length bitstrings, etc.

1 Counting Cartesian Products

Note 11

For two sets A and B, define the cartesian product as $A \times B = \{(a,b) : a \in A, b \in B\}$.

(a) Given two countable sets A and B, prove that $A \times B$ is countable.

(b) Given a finite number of countable sets $A_1, A_2, \ldots, A_n$, prove that

$$A_1 \times A_2 \times \cdots \times A_n$$

is countable.

(c) Consider a countably infinite number of finite sets: $B_1, B_2, \ldots$ for which each set has at least 2 elements. Prove that $B_1 \times B_2 \times \cdots$ is uncountable.

Solution:

(a) As shown in lecture, $\mathbb{N} \times \mathbb{N}$ is countable by creating a zigzag map that enumerates through the pairs: $(0,0), (1,0), (0,1), (2,0), (1,1), \ldots$. Since A and B are both countable, there exists a bijection between each set and a subset of $\mathbb{N}$. Thus we know that $A \times B$ is countable because there is a bijection between a subset of $\mathbb{N} \times \mathbb{N}$ and $A \times B : f(i,j) = (A_i, B_j)$. We can enumerate the pairs (a,b) similarly.

(b) Proceed by induction.
Base Case: $n = 2$. We showed in part (a) that $A_1 \times A_2$ is countable since both A_1 and A_2 are countable.
Induction Hypothesis: Assume that for some $n \in \mathbb{N}$, $A_1 \times A_2 \times \cdots \times A_n$ is countable.
Induction Step: Consider $A_1 \times \cdots \times A_n \times A_{n+1}$. We know from our hypothesis that $A_1 \times \cdots \times A_n$ is countable, call it $C = A_1 \times \cdots \times A_n$. We proved in part (a) that since C is countable and A_{n+1} are countable, $C \times A_{n+1}$ is countable, which proves our claim.

(c) Let us assume that each B_i has size 2. If any of the sizes are greater than 2, that would only make the cartesian product larger. Notice that this is equivalent to the set of infinite length binary strings, which was proven to be uncountable in the notes.

Alternatively, we could provide a diagonalization argument: Assuming for the sake of contradiction that $B_1 \times B_2 \times \cdots$ is countable and its elements can be enumerated in a list:

$$
\begin{aligned}
&(b_{1,1}, b_{2,1}, b_{3,1}, b_{4,1}, \dots)\\
&(b_{1,2}, b_{2,2}, b_{3,2}, b_{4,2}, \dots)\\
&(b_{1,3}, b_{2,3}, b_{3,3}, b_{4,3}, \dots)\\
&(b_{1,4}, b_{2,4}, b_{3,4}, b_{4,4}, \dots)\\
&\vdots
\end{aligned}
$$

where $b_{i,j}$ represents the item from set B_i that is included in the jth element of the Cartesian Product. Now consider the element $(\overline{b_{1,1}}, \overline{b_{2,2}}, \overline{b_{3,3}}, \overline{b_{4,4}}, \dots)$, where $\overline{b_{i,j}}$ represents any item from set B_i that differs from $b_{i,j}$ (i.e. any other element in the set). This is a valid element that should exist in the Cartesian Product $B_1, B_2, \dots$, yet it is not in the enumerated list. This is a contradiction, so $B_1 \times B_2 \times \cdots$ must be uncountable.

2 Computability Intro

Note 12

Computability: The main focus is on the Halting problem, and programs that provably cannot exist.

The *Halting problem* is the problem of determining whether a program P run on input x ever halts, or whether it loops forever. It turns out that there does not exist any program that solves this problem.

Using this information, we can prove that other problems also cannot be solved by a computer program, through the use of *reductions*. The main idea is to show that if a given problem can be solved by a computer program `TestX`, then the Halting problem can also be solved by a computer program `TestHalt` that uses `TestX` as a subroutine.

The primary template we'll use for this course is as follows. Suppose we want to show that a program `TestX` does not exist, where `TestX(Q, y)` tries to determine whether a program Q on input y does some task $\mathscr{X}$ (i.e. it outputs "True" if $Q(y)$ does the task $\mathscr{X}$, and it outputs "False" if $Q(y)$ does not do the task $\mathscr{X}$). We can define `TestHalt` as follows (in pseudocode):

```
def TestHalt(P, x):
    def Q(y):
        run P(x)
        do 𝒳
    return TestX(Q, y)  # for some given y
```

Note that this template will be sufficient for our purposes in CS70, but more complex reductions will require more sophisticated programs—you'll learn more about this in classes like CS170 and CS172.

(a) Consider the reduction template given above. Let's break down what it's doing.

We follow an argument by contradiction—we assume that there is a program `TestX(Q, y)` that is able to determine whether another program Q on input y does some task $\mathscr{X}$.

There are two cases: either $P(x)$ halts, or it loops forever. We'd like to show that `TestHalt` as defined above returns the correct answer in both of these cases.

(i) Suppose $P(x)$ halts. What does `TestHalt` return, and why?

(ii) Suppose $P(x)$ loops forever. What does `TestHalt` return, and why?

(iii) What does this tell us about the existence of `TestX`? Briefly justify your answer.

Solution:

(a) (i) If $P(x)$ halts, then $Q(y)$ will finish executing $P(x)$, and eventually do the task $\mathscr{X}$. This means that `TestX` would return "True", since $Q(y)$ does eventually do $\mathscr{X}$.

(ii) If $P(x)$ loops forever, then $Q(y)$ will get stuck while executing $P(x)$, and will never get to doing the task $\mathscr{X}$. This means that `TestX` would return "False", since $Q(y)$ never does $\mathscr{X}$.

(iii) These answers returned by `TestHalt` exactly solve the Halting problem! However, we've already shown that the Halting problem cannot be solved by a computer program—this is a contradiction. As such, `TestX` cannot exist.

3 Hello World!

Note 12

Determine the computability of the following tasks. If it's not computable, write a reduction or self-reference proof. If it is, write the program. Throughout this problem, you are allowed to execute programs while surpressing their print statements.

(a) You want to determine whether a program P on input x prints "Hello World!". Is there a computer program that can perform this task? Justify your answer.

(b) You want to determine whether a program P prints "Hello World!" while or before running the kth line in the program. Is there a computer program that can perform this task? Justify your answer.

(c) You want to determine whether a program P prints "Hello World!" in the first k steps of its execution. Is there a computer program that can perform this task? Justify your answer.

Solution:

(a) Uncomputable. We will reduce `TestHalt` to `PrintsHW`(P,x).

```
TestHalt(P, x):
  P'(x):
    run P(x) while suppressing print statements
    print("Hello World!")

  return PrintsHW(P', x)
```

If `PrintsHW` exists, `TestHalt` must also exist by this reduction. Since `TestHalt` cannot exist, `PrintsHW` cannot exist.

(b) Uncomputable. We will reduce `TestHalt` to `PrintsHWByK`(P,x,k).

```
TestHalt(P, x):
    P'(x):
        run P(x) while suppressing print statements
        print("Hello World!")
    return PrintsHWByK(P', x, 2)
```

Here, we notice that `P'` has only two lines (or at most `len(P) + 1` lines, depending on how this is implemented), and we print "Hello World!" by the last line of `P'` if and only if `P(x)` halts.

Alternatively, we can reduce `PrintsHW`(P,x) from part (a) to this program `PrintsHWByK`(P,x,k):

```
PrintsHW(P, x):
  for i in range(len(P)):
    if PrintsHWByK(P, x, i):
      return true
  return false
```

Note that we technically need to iterate through all the lines here, since there may be large jumps within the code of `P`; this means that we may for example jump from line 1 to line 100 and back to line 2 to print "Hello World!", but `PrintsHWByK`$(P,x,100)$ will return false, since we first reach line 100 without printing "Hello World!".

(c) Computable. You can simply run the program until k steps are executed. If P has printed "Hello World!" by then, return true. Else, return false.

The reason that part (b) is uncomputable while part (c) is computable is that it's not possible to determine if we ever execute a specific line because this depends on the logic of the program, but the number of computer instructions can be counted.

4 Code Reachability

Note 12

Consider triplets (M, x, L) where

- `M` is a Java program
- `x` is some input
- `L` is an integer

and the question of: if we execute $M(x)$, do we ever hit line L?

Prove this problem is undecidable.

Solution: Suppose we had a procedure that could decide the above; call it `Reachable(M, x, L)`. Consider the following example of a program deciding whether $P(x)$ halts:

```
def Halt(P, x):
    def M(t):
        run P(x)   # line 1 of M
        return     # line 2 of M
    return Reachable(M, 0, 2)
```

Program M reaches line 2 if and only if $P(x)$ halted. Thus, we have implemented a solution to the halting problem — contradiction.