

1 Countability Intro

Note 12

A function $f : A \rightarrow B$ maps elements from set A to set B .

f is *injective* if it maps distinct elements to distinct elements, and *surjective* if every element in B is mapped to by some element in A . If f is both injective and surjective, it is *bijective*, and the sets A and B are said to have the same *cardinality* (size). The cardinality of a set is denoted by $|A|$.

f is bijective if and only if there exists an inverse function $f^{-1} : B \rightarrow A$ such that $f^{-1}(f(a)) = a$ for all $a \in A$ and $f(f^{-1}(b)) = b$ for all $b \in B$.

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, $f : S \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow S$. Intuitively, this is because an injection $f : S \rightarrow \mathbb{N}$ means $|S| \leq |\mathbb{N}|$, and an injection $g : \mathbb{N} \rightarrow S$ means $|\mathbb{N}| \leq |S|$; together, we have $|\mathbb{N}| = |S|$.

- *Uncountably infinite:* a set with a cardinality larger than $|\mathbb{N}|$. Therefore, there is no bijection with $\mathbb{N}$ and the set can't be listed out.

Use *Cantor diagonalization* to prove uncountability through contradiction; the classic example is the set of reals in $[0, 1]$:

$\mathbb{N}$	$[0, 1]$
0	0 . 7 3 2 0 5 0 ...
1	0 . 4 1 4 2 1 3 ...
2	0 . 6 1 8 0 3 3 ...
3	0 . 1 8 2 8 1 8 ...
4	0 . 1 4 1 5 9 2 ...
5	0 . 5 7 7 2 1 5 ...
$\vdots$	$\vdots$
?	0 . 8 2 9 9 1 6 ...

To have a bijection with $\mathbb{N}$, we must be able to assign a natural number to every real number, as above. In other words, every real number must eventually appear on this (infinite) list. However, if we change the digits along the diagonal and compile that into a new real number, it is different from every single element in the list in at least one place, so it's not in the list—this is a **contradiction**, and it implies that there is at least one real number “left over”. Hence, the cardinality of $\mathbb{R}$ is strictly greater than $\mathbb{N}$!

Sometimes it can be easier to prove countability (or uncountability) through bijections with other countable (or 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.

- (a) Read through the previous page's description of Cantor diagonalization. In your own words, why do we know that the number created by changing the digits along the diagonal is not in the list?
- (b) True or false: the new number created is a real number in $[0, 1]$ that is not on the list.
- (c) Your friend is confused about how Cantor diagonalization doesn't apply to the set of natural numbers. They argue that natural numbers can be thought of as an infinite length string of digits, by padding each number with an infinite number of zeroes to the left. If we then assume by contradiction that we can list out the set of natural numbers with the padded 0's, we can change the digits along a diagonal, to create a new natural number not in the list.

0	...	0	0	0	0	①
1	...	0	1	2	③	4
2	...	5	2	⑧	2	3
3	...	9	④	3	2	1
⋮				⋮		
?	...	1	5	9	4	2

What is wrong with this argument?

Solution:

- (a) The newly created number will differ with every number in the list by at least one digit.
- (b) True. Observe that it is of the form 0.(decimal), which is a real number and in $[0, 1]$. Moreover, this number cannot possibly be on the list since it differs with every number on the list by at least one digit.
- (c) The issue here is that the newly created number is not necessarily a natural number. This number has an infinite number of digits (we'll always be changing the padded zeroes into some nonzero digits), while natural numbers must have a finite number of digits.

This means that when you perform Cantor diagonalization to show a set S is uncountable, it is imperative that you ensure that the newly created element is always still an element of S ; otherwise, we cannot say anything about the result of the diagonalization.

2 Counting Cartesian Products

Note 12

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 $A_1, A_2, \dots, A_n$, prove that

$$A_1 \times A_2 \times \dots \times A_n$$

is countable.

- (c) Consider a countably infinite number of finite sets: $B_1, B_2, \dots$ for which each set has at least 2 elements. Prove that $B_1 \times B_2 \times \dots$ 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: $(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), \dots$. 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 j th 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.

3 Counting Functions

Note 12

Are the following sets countable or uncountable? Prove your claims.

- (a) The set of all functions f from $\mathbb{N}$ to $\mathbb{N}$ such that f is non-decreasing. That is, $f(x) \leq f(y)$ whenever $x \leq y$.
- (b) The set of all functions f from $\mathbb{N}$ to $\mathbb{N}$ such that f is non-increasing. That is, $f(x) \geq f(y)$ whenever $x \leq y$.

Solution:

- (a) Uncountable: Let us assume the contrary and proceed with a diagonalization argument. If there are countably many such functions, we can enumerate them as

	0	1	2	3	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Now go along the diagonal and define f such that $f(x) > f_x(x)$ and $f(y) = f(x)$ if $y > x$, which is possible because at step k we only need to find a number $\in \mathbb{N}$ greater than all the $f_j(j)$ for $j \in \{0, 1, \dots, k\}$.

$\{0, \dots, k\}$; for example, we could define such a function using

$$f(x) = \begin{cases} f_0(0) + 1 & x = 0 \\ \max(f_x(x), f(x-1)) + 1 & x > 0 \end{cases}$$

This function differs from each f_i and therefore cannot be on the list, hence the list does not exhaust all non-decreasing functions. As a result, there must be uncountably many such functions.

Alternative Solution: Look at the subset $\mathcal{S}$ of strictly increasing functions. Any such f is uniquely identified by its image which is an infinite subset of $\mathbb{N}$. But the set of infinite subsets of $\mathbb{N}$ is uncountable. This is because the set of all subsets of $\mathbb{N}$ is uncountable, and the set of all finite subsets of $\mathbb{N}$ is countable. So $\mathcal{S}$ is uncountable and hence the set of all non-decreasing functions must be too.

Alternative Solution 2: We can inject the set of infinitely long binary strings into the set of non-decreasing functions as follows. For any infinitely long binary string b , let $f(n)$ be equal to the number of 1's appearing in the first n -digits of b . It is clear that the function f so defined is non-decreasing. Also, since the function f is uniquely defined by the infinitely long binary string, the mapping from binary strings to non-decreasing functions is injective. Since the set of infinite binary strings is uncountable, and we produced an injection from that set to the set of non-decreasing functions, that set must be uncountable as well.

- (b) Countable. Since $f(0) \geq f(1) \geq f(2) \geq \dots \geq 0$ and the values are natural numbers, f can decrease at most $f(0)$ times, so it is eventually constant: there is a smallest m such that $f(x) = f(m)$ for all $x \geq m$. Then f is completely determined by the finite list $(f(0), f(1), \dots, f(m))$, and different functions give different lists. This is an injection from the set of non-increasing functions into the set of finite lists of natural numbers.

The set of finite lists of natural numbers is countable: write each list as a string over the 11-symbol alphabet $\{0, 1, \dots, 9, ,\}$ (for example, $(12, 3, 3, 0)$ becomes $12,3,3,0$). There are only finitely many strings of each length, so we can enumerate all strings of length 1, then all of length 2, and so on. Since the non-increasing functions inject into a countable set, they are countable.

4 Computability Intro

Note 13

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

The *Halting problem* is the problem of determining whether **any arbitrary** 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.

Here is an example of a template that we will often use for reductions. Suppose we want to show that a program `TestX` does not exist, where `TestX(P', y)` tries to determine whether a program P' on input y does some task $\mathcal{X}$ (i.e. it outputs “True” if $P'(y)$ does the task $\mathcal{X}$, and it outputs “False” if $P'(y)$ does not do the task $\mathcal{X}$). We can define `TestHalt` as follows (in pseudocode):

```
def TestHalt(P, x):  
    def P'(y):  
        run P(x)  
        do  $\mathcal{X}$   
    return TestX(P', y) # for some given y
```

Then we will show that if `TestX` exists, then `TestHalt` would correctly solve the halting problem.

Note that reductions in CS70 will generally look like this template. 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(P', y)` that is able to determine whether another program P' on input y does some task $\mathcal{X}$.

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 $P'(y)$ will finish executing $P(x)$, and eventually do the task $\mathcal{X}$. This means that `TestX` would return “True”, since $P'(y)$ does eventually do $\mathcal{X}$.
- (ii) If $P(x)$ loops forever, then $P'(y)$ will get stuck while executing $P(x)$, and will never get to doing the task $\mathcal{X}$. This means that `TestX` would return “False”, since $P'(y)$ never does $\mathcal{X}$.
- (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. Note that y is never explicitly used in the function P' in this problem. It is a placeholder for the template, as other reductions may want to use it.

5 Hello World!

Note 13

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

Here, we notice that P' has only two lines (or at most $\text{len}(P) + 1$ 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.

6 Code Reachability

Note 13

Consider triplets (M, x, L) where

- M is a 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 that this problem is uncomputable.

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.