EECS 70 Discrete Mathematics and Probability Theory
Spring 2020 Note 12

To Infinity And Beyond: Countability and Computability

This note ties together two topics that might seem like they have nothing to do with each other. The nature of infinity (and more particularly, the distinction between different levels of infinity) and the fundamental nature of computation and proof. This note can only scratch the surface — if you want to understand this material more deeply, there are wonderful courses in the Math department as well as CS172 and graduate courses like EECS229A that will connect this material to the nature of information and compression as well.

1 Cardinality: Infinity and Countability

In this note we'll first discuss the question of when two sets have the same cardinality, or size. This is a simple issue for finite sets, but for infinite sets it becomes subtle. We'll see how to formulate the question precisely, and then see several quite surprising consequences. To set the scene, we begin with the simple case of finite sets.

Bijections

Two finite sets have the same size if and only if their elements can be paired up, so that each element of one set has a unique partner in the other set, and vice versa. We formalize this through the concept of a *bijection*.

Consider a function[1] (or mapping) f that maps elements of a set A (called the *domain* of f) to elements of set B (called the *range* of f). Since f is a function, it must specify, for each element $x \in A$ ("input"), exactly one element $f(x) \in B$ ("output"). Recall that we write this as $f : A \to B$. We say that f is a *bijection* if every element $a \in A$ has a unique *image* $b = f(a) \in B$, and every element $b \in B$ has a unique *pre-image* $a \in A$ such that $f(a) = b$.

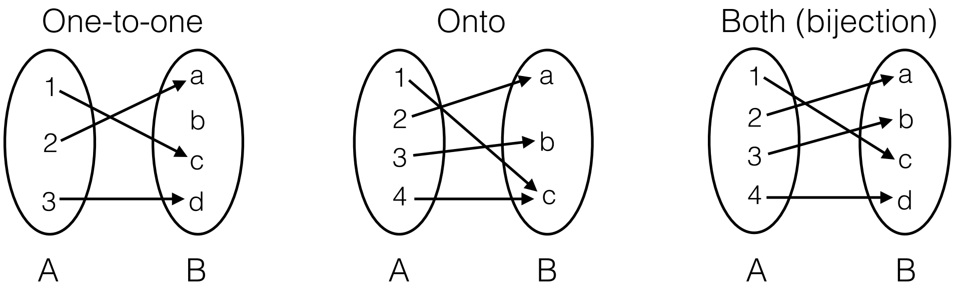
f is a *one-to-one function* (or an *injection*) if f maps distinct inputs to distinct outputs. More rigorously, f is one-to-one if the following holds: $x \neq y \Rightarrow f(x) \neq f(y)$.

f is *onto* (or *surjective*) if it "hits" every element in the range (i.e., each element in the range has at least one pre-image). More precisely, a function f is onto if the following holds: $(\forall y\ \exists x)(f(x) = y)$.

Here are some simple examples to help visualize one-to-one and onto functions, and bijections:

Note that, according to the above definitions, f is a bijection if and only if it is both one-to-one and onto.

[1] See Note 0 for a review of basic definitions connected with functions.



Exercise. Let $f : A \to B$ be a bijection. Show that f has an *inverse* $f^{-1} : B \to A$ that satisfies $f^{-1}(f(a)) = a$ for all $a \in A$, and that f^{-1} is also a bijection.

Cardinality

How can we determine whether two sets have the same *cardinality* (or "size")? The answer to this question, reassuringly, lies in early grade school memories: by demonstrating a *pairing* between elements of the two sets. More formally, we need to demonstrate a *bijection* f between the two sets. The bijection sets up a one-to-one correspondence, or pairing, between elements of the two sets. We've seen above how this works for finite sets. In the rest of this lecture, we will see what it tells us about *infinite* sets.

Our first question about infinite sets is the following: Are there more natural numbers $\mathbb{N}$ than there are positive integers $\mathbb{Z}^+$? It is tempting to answer yes, since every positive integer is also a natural number, but the natural numbers have one extra element $0 \notin \mathbb{Z}^+$. Upon more careful observation, though, we see that we can define a mapping between the natural numbers and the positive integers as follows:

$$\begin{array}{lccccccccc}
\mathbb{N} & 0 & 1 & 2 & 3 & 4 & 5 & \dots & & \\
f \downarrow & & \searrow & \searrow & \searrow & \searrow & \searrow & \searrow & & \\
\mathbb{Z}^+ & & 1 & 2 & 3 & 4 & 5 & 6 & \dots &
\end{array}$$

Why is this mapping a bijection? Clearly, the function $f : \mathbb{N} \to \mathbb{Z}^+$ is onto because every positive integer is hit. And it is also one-to-one because no two natural numbers have the same image. (The image of n is $f(n) = n+1$, so if $f(n) = f(m)$ then we must have $n = m$.) Since we have shown a bijection between $\mathbb{N}$ and $\mathbb{Z}^+$, this tells us that there are exactly as many natural numbers as there are positive integers! (Very) informally, we have proved that "$\infty + 1 = \infty$."

What about the set of *even* natural numbers $2\mathbb{N} = \{0, 2, 4, 6, ...\}$? In the previous example the difference was just one element. But in this example, there seem to be twice as many natural numbers as there are even natural numbers. Surely, the cardinality of $\mathbb{N}$ must be larger than that of $2\mathbb{N}$ since $\mathbb{N}$ contains all of the odd natural numbers as well? Though it might seem to be a more difficult task, let us attempt to find a bijection between the two sets using the following mapping:

$$\begin{array}{lcccccccc}
\mathbb{N} & & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\
f\downarrow & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
2\mathbb{N} & & 0 & 2 & 4 & 6 & 8 & 10 & \dots
\end{array}$$

The mapping in this example is also a bijection. f is clearly one-to-one, since distinct natural numbers get mapped to distinct even natural numbers (because $f(n) = 2n$). f is also onto, since every n in the range is hit: its pre-image is $\frac{n}{2}$. Since we have found a bijection between these two sets, this tells us that in fact $\mathbb{N}$ and $2\mathbb{N}$ have the same cardinality!

What about the set of all integers, $\mathbb{Z}$? At first glance, it may seem obvious that the set of integers is larger than the set of natural numbers, since it includes infinitely many negative numbers. However, as it turns out, it is possible to find a bijection between the two sets, meaning that the two sets have the same size! Consider the following mapping f:

$$0 \to 0,\ 1 \to -1,\ 2 \to 1,\ 3 \to -2,\ 4 \to 2,\ \dots,\ 124 \to 62,\ \dots$$

In other words, our function is defined as follows:

$$f(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ \frac{-(x+1)}{2}, & \text{if } x \text{ is odd} \end{cases}$$

We will prove that this function $f : \mathbb{N} \to \mathbb{Z}$ is a bijection, by first showing that it is one-to-one and then showing that it is onto.

Proof (one-to-one): Suppose $f(x) = f(y)$. Then they both must have the same sign. Therefore either $f(x) = \frac{x}{2}$ and $f(y) = \frac{y}{2}$, or $f(x) = \frac{-(x+1)}{2}$ and $f(y) = \frac{-(y+1)}{2}$. In the first case, $f(x) = f(y) \Rightarrow \frac{x}{2} = \frac{y}{2} \Rightarrow x = y$. Hence $x = y$. In the second case, $f(x) = f(y) \Rightarrow \frac{-(x+1)}{2} = \frac{-(y+1)}{2} \Rightarrow x = y$. So in both cases $f(x) = f(y) \Rightarrow x = y$, so f is injective.

Proof (onto): If $y \in \mathbb{Z}$ is non-negative, then $f(2y) = y$. Therefore, y has a pre-image. If y is negative, then $f(-(2y+1)) = y$. Therefore, y has a pre-image. Thus every $y \in \mathbb{Z}$ has a preimage, so f is onto.

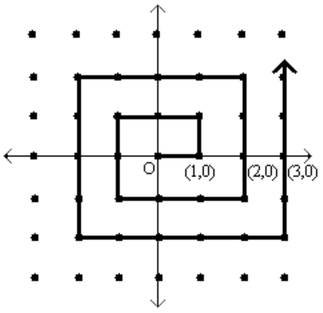
Since f is a bijection, this tells us that $\mathbb{N}$ and $\mathbb{Z}$ have the same size.

Now for an important definition. We say that a set S is **countable** if there is a bijection between S and $\mathbb{N}$ or some subset of $\mathbb{N}$. Thus any finite set S is countable (since there is a bijection between S and the subset $\{0, 1, 2, \dots, m-1\}$, where $m = |S|$ is the size of S). And we have already seen three examples of countable infinite sets: $\mathbb{Z}^+$ and $2\mathbb{N}$ are obviously countable since they are themselves subsets of $\mathbb{N}$; and $\mathbb{Z}$ is countable because we have just seen a bijection between it and $\mathbb{N}$.

What about the set of all rational numbers? Recall that $\mathbb{Q} = \{\frac{x}{y} \mid x, y \in \mathbb{Z},\ y \neq 0\}$. Surely there are more rational numbers than natural numbers? After all, there are infinitely many rational numbers between any two natural numbers. Surprisingly, the two sets have the same cardinality! To see this, let us introduce a slightly different way of comparing the cardinality of two sets.

If there is a one-to-one function $f : A \to B$, then the cardinality of A is less than or equal to that of B. Now to show that the cardinality of A and B are the same we can show that $|A| \leq |B|$ and $|B| \leq |A|$. This corresponds to showing that there is a one-to-one function $f : A \to B$ and a one-to-one function $g : B \to A$. The existence of these two one-to-one functions implies that there is a bijection $h : A \to B$, thus showing that A and B have the same cardinality. The proof of this fact, which is called the Cantor-Schröder-Bernstein theorem, is actually interesting, and we will skip it here — instead walking you through it on the homework. For us, this fact will be very useful: for example, to show that a set S is countable, it is enough to give separate injections $f : S \to \mathbb{N}$ and $g : \mathbb{N} \to S$, rather than designing a bijection (which is often trickier).

Back to comparing the natural numbers and the integers. First it is obvious that $|\mathbb{N}| \leq |\mathbb{Q}|$ because $\mathbb{N} \subseteq \mathbb{Q}$. So our goal now is to prove that also $|\mathbb{Q}| \leq |\mathbb{N}|$. To do this, we must exhibit an injection $f : \mathbb{Q} \to \mathbb{N}$. The following picture of a spiral conveys the idea of this injection:



Each rational number $\frac{a}{b}$ (written in its lowest terms, so that $\gcd(a,b) = 1$) is represented by the point (a,b) in the infinite two-dimensional grid shown (which corresponds to $\mathbb{Z} \times \mathbb{Z}$, the set of all pairs of integers). Note that not all points on the grid are valid representations of rationals: e.g., all points on the x-axis have $b = 0$ so none are valid (except for $(0,0)$, which we take to represent the rational number 0); and points such as $(2,8)$ and $(-1,-4)$ are not valid either as the rational number $\frac{1}{4}$ is represented by $(1,4)$. But $\mathbb{Z} \times \mathbb{Z}$ certainly contains all rationals under this representation, so if we come up with an injection from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{N}$ then this will also be an injection from $\mathbb{Q}$ to $\mathbb{N}$ (why?).

The idea is to map each pair (a,b) to its position along the spiral, starting at the origin. (Thus, e.g.,$(0,0) \to 0$, $(1,0) \to 1$, $(1,1) \to 2$, $(0,1) \to 3$, and so on.) It should be clear that this mapping maps every pair of integers injectively to a natural number, because each pair occupies a unique position along the spiral.

This tells us that $|\mathbb{Q}| \leq |\mathbb{N}|$. Since also $|\mathbb{N}| \leq |\mathbb{Q}|$, as we observed earlier, by the Cantor-Schröder-Bernstein Theorem $\mathbb{N}$ and $\mathbb{Q}$ have the same cardinality.

Exercise. Show that the set $\mathbb{N} \times \mathbb{N}$ of all ordered pairs of natural numbers is countable.

Our next example concerns the set of all binary strings (of any finite length), denoted $\{0,1\}^*$. Despite the fact that this set contains strings of unbounded length, it turns out to have the same cardinality as $\mathbb{N}$. To see this, we set up a direct bijection $f : \mathbb{N} \to \{0,1\}^*$ as follows. Note that it suffices to *enumerate* the elements of $\{0,1\}^*$ in such a way that each string appears exactly once in the list. We then get our bijection by setting $f(n)$ to be the nth string in the list. How do we enumerate the strings in $\{0,1\}^*$? Well, it's natural to list them in increasing order of length, and then (say) in *lexicographic* order (or, equivalently, numerically increasing order when viewed as binary numbers) within the strings of each length. This means that the list would look like

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 1000, \ldots,$$

where ε denotes the empty string (the only string of length 0). It should be clear that this list contains each binary string once and only once, so we get a bijection with $\mathbb{N}$ as desired.

Our final countable example is the set of all polynomials with natural number coefficients, which we denote $\mathbb{N}(x)$. To see that this set is countable, we give injections from $\mathbb{N}$ to $\mathbb{N}(x)$ and from $\mathbb{N}(x)$ to $\mathbb{N}$. The first of these is easy, since each natural number n is itself already trivially a polynomial. For the injection the other way, from $\mathbb{N}(x)$ to $\mathbb{N}$, we will make use of (a variant of) the previous example. Note first that, by essentially the same argument as for $\{0,1\}^*$, we can see that the set of all *ternary* strings $\{0,1,2\}^*$ (that is, strings over the alphabet $\{0,1,2\}$) is countable. It therefore suffices to exhibit an injection $f : \mathbb{N}(x) \to \{0,1,2\}^*$, which in turn will give an injection from $\mathbb{N}(x)$ to $\mathbb{N}$.

How do we define f? Let's first consider an example, namely the polynomial $p(x) = 5x^5 + 2x^4 + 7x^3 + 4x + 6$. We can list the coefficients of $p(x)$ as follows: $(5,2,7,0,4,6)$. We can then write these coefficients as binary strings: $(101,10,111,0,100,110)$. Now, we can construct a ternary string where a “2" is inserted as a separator between each binary coefficient (ignoring coefficients that are 0). Thus we map $p(x)$ to a ternary string as illustrated below:

$$\begin{tikzcd} 5x^5 + 2x^4 + 7x^3 + 4x + 6 \arrow[d] \\ 1012102111221002110 \end{tikzcd}$$

It is easy to check that this is an injection, since the original polynomial can be uniquely recovered from this ternary string by simply reading off the coefficients between each successive pair of 2's. (Notice that this mapping $f : \mathbb{N}(x) \to \{0,1,2\}^*$ is not onto (and hence not a bijection) since many ternary strings will not be the image of any polynomials; this will be the case, for example, for any ternary strings that contain binary subsequences with leading zeros.)

Hence we have an injection from $\mathbb{N}(x)$ to $\mathbb{N}$, and from $\mathbb{N}$ to $\mathbb{N}(x)$, so $\mathbb{N}(x)$ is countable.

Cantor's Diagonalization

We have established that $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ all have the same cardinality. What about $\mathbb{R}$, the set of real numbers? Surely they are countable too? After all, the rational numbers, like the real numbers, are dense (i.e., between any two rational numbers a,b there is a rational number, namely $\frac{a+b}{2}$). In fact, between any two *real* numbers there is always a rational number. It is really surprising, then, that there are strictly more real numbers than rationals! That is, there is *no* bijection between the rationals (or the natural numbers) and the reals. We shall now prove this, using a beautiful argument due to Cantor that is known as *diagonalization.* In fact, we will show something even stronger: the real numbers in the interval $[0,1]$ are uncountable!

Exercise. Show how to find a rational number between any two (distinct) real numbers.

In preparation for the proof, recall that any real number can be written out uniquely as an infinite decimal with no trailing zeros. In particular, a real number in the interval $[0,1]$ can be written as $0.d_1d_2d_3\ldots$. In this representation, we write for example[2] 1 as $0.999\ldots$, and 0.5 as $0.4999\ldots$. (Thus rational numbers will always be represented as recurring decimals, while irrational ones will be represented as non-recurring ones. Importantly for us, all of these expressions will be infinitely long and unique.)

[2] To see this, write $x = .999\ldots$. Then $10x = 9.999\ldots$, so $9x = 9$, and thus $x = 1$.

Theorem: The real interval $\mathbb{R}[0,1]$ (and hence also the set of real numbers $\mathbb{R}$) is uncountable.

Proof: Suppose towards a contradiction that there is a bijection $f : \mathbb{N} \to \mathbb{R}[0,1]$. Then, we can enumerate the real numbers in an infinite list $f(0), f(1), f(2), \ldots$ as follows:

$$\begin{array}{l}
f(0) = 0\ .\textcircled{5}2\ 1\ 4\ 9\ 3\ 5\ 6\ \ldots \\
f(1) = 0\ .\ 1\textcircled{4}1\ 6\ 2\ 9\ 8\ 5\ \ldots \\
f(2) = 0\ .\ 9\ 4\textcircled{7}8\ 2\ 7\ 1\ 2\ \ldots \\
f(3) = 0\ .\ 5\ 3\ 0\textcircled{9}8\ 1\ 7\ 5\ \ldots \\
\ \vdots \qquad\qquad\qquad\quad \vdots
\end{array}$$

Note that we have circled the digits on the *diagonal* of this list. This sequence of circled digits can be viewed as a real number, $r = 0.5479\ldots$, since it is an infinite decimal expansion.

Now consider the real number s obtained by modifying every digit of r, say by replacing each digit d with $d+1 \pmod{10}$ if $d \neq 9$, and with[3] 1 if $d = 9$; thus in our example above, $s = 0.6581\ldots$. We claim that s does not occur in our infinite list of real numbers. Suppose that it did, and that it was the n^{th} number in the list, $f(n)$. But by construction s differs from $f(n)$ in the $(n+1)$th digit, so these two numbers cannot be equal! So we have constructed a real number s that is not in the range of f. But this contradicts our original assertion that f is a bijection. Hence the real numbers are not countable.

It is worth asking what happens if we apply the same method to $\mathbb{Q}$, in a (presumably futile) attempt to show that the rationals are uncountable. Well, suppose for contradiction that our bijective function $f : \mathbb{N} \to \mathbb{Q}[0,1]$ produces the following mapping:

$$\begin{array}{l}
f(0) = 0\ .\textcircled{1}4\ 0\ 0\ 0\ \ldots \\
f(1) = 0\ .\ 5\textcircled{9}2\ 4\ 5\ \ldots \\
f(2) = 0\ .\ 2\ 1\textcircled{4}2\ 1\ \ldots \\
\ \vdots \qquad\qquad\quad \vdots
\end{array}$$

Again, we consider the number q obtained by modifying every digit of the diagonal as before, giving $q = 0.215\ldots$ in the above example. Again, by construction, the number q does not appear in the list. However, this tells us nothing because we do not know that q is rational (indeed, it is extremely unlikely for the decimal expansion to be periodic, as required for q to be rational); hence the fact that q is not in the list of rationals is *not* a contradiction! When dealing with the reals, the modified diagonal number was guaranteed to be a real number.

2 Self-Reference and Computability

Cantor's diagonalization argument turns out to be applicable far more generally than just for establishing the uncountability of the real numbers. (This once again strengthens the general EECS perspective towards mathematical thinking — the underlying concepts and proofs are almost always more important to us than the theorems.) However, one additional critical ingredient turns out to be required: the notion of "self-reference" (having a statement or program somehow be about itself). This has far-reaching consequences for the limits of computation (the Halting Problem) and the foundations of logic in mathematics (Gödel's

[3] The reason we treat $d = 9$ differently is that the same real number can have two decimal expansions; e.g., $0.999... = 1.000$.

incompleteness theorem). Extremely recently, this has also turned out to have important consequences in physics due to the power of certain models of quantum computation.

The Liar's Paradox

Recall that propositions are statements that are either true or false. We saw in an earlier lecture that some statements are not well defined or too imprecise to be called propositions. But here is a statement that is problematic for more subtle reasons:

"All Cretans are liars."

So said a Cretan in antiquity, thus giving rise to the so-called liar's paradox which has amused and confounded people over the centuries. Why? Because if the statement above is true, then the Cretan was lying, which implies the statement is false. But actually the above statement isn't really a paradox; it simply yields a contradiction if we assume it is true, but if it is false then there is no problem.

A true formulation of this paradox is the following statement:

"This statement is false."

Is the statement above true? If the statement is true, then what it asserts must be true; namely that it is false. But if it is false, then it must be true. So it really is a paradox, and we see that it arises because of the self-referential nature of the statement. Around a century ago, this paradox found itself at the center of foundational questions about mathematics and computation.

We will now study how this paradox relates to computation. Before doing so, let us consider another manifestation of the paradox, attributed to the great logician Bertrand Russell (but actually a version of a related paradox devised by Russell). In a village with just one barber, every man keeps himself clean-shaven. Some of the men shave themselves, while others go to the barber. The barber proclaims:

"I shave all and only those men who do not shave themselves."

It seems reasonable then to ask the question: Does the barber shave himself? Thinking more carefully about the question though, we see that, assuming that the barber's statement is true, we are presented with the same self-referential paradox: a logically impossible scenario. If the barber does not shave himself, then according to what he announced, he shaves himself. If the barber does shave himself, then according to his statement he does not shave himself!

(Of course, the real resolution to the barber paradox is simple: the barber is a woman who doesn't need to shave. But that's not the point.)

2.1 The Halting Problem

Are there tasks that a computer cannot perform? For example, we would like to ask the following basic question when compiling a program: does it run forever, i.e. go into what feels like an infinite loop? In 1936, Alan Turing showed that there is no program that can perform this test. The proof of this remarkable fact is very elegant and combines two ingredients: self-reference (as in the liar's paradox), and the fact that we cannot separate programs from data. In computers, a program is represented by a finite string of bits just as integers, characters, and other data are. The only difference is in how the string of bits is interpreted.

We will now examine the Halting Problem. Given the description of a program and its input, we would like to know if the program ever halts when it is executed on the given input. In other words, we would like to write a program `TestHalt` that behaves as follows:

$$\texttt{TestHalt(P,x)} = \begin{cases} \text{“yes''}, & \text{if program } P \text{ halts on input } x \\ \text{“no''}, & \text{if program } P \text{ loops on input } x \end{cases}$$

Why can't such a program exist? First, let us use the fact that a program is just a bit string, so it can be input as data. This means that it is perfectly valid to consider the behavior of `TestHalt(P,P)`, which will output “yes” if P halts on P, and “no” if P loops forever on P. We now prove that such a program cannot exist.

Theorem: *The Halting Problem is uncomputable; i.e., there does not exist a computer program TestHalt with the behavior specified above on all inputs (P,x). (Note that this statement holds regardless of what hardware or programming language we use.)*

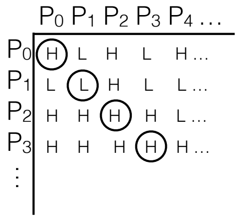
Proof: Assume for contradiction that the program `TestHalt` exists. Then we can easily use it as a subroutine to construct the following program:

```
Turing(P)

  if TestHalt(P,P) = "yes" then loop forever

  else halt
```

So if the program P when given P as input halts, then `Turing(P)` loops forever; otherwise, `Turing(P)` halts. Note that the program `Turing` is very easy to construct if we are given the program `TestHalt`.

Now let us look at the behavior of `Turing(Turing)`. There are two cases: either it halts, or it does not. If `Turing(Turing)` halts, then it must be the case that `TestHalt(Turing,Turing)` returned “no.” But by definition of `TestHalt`, that would mean that `Turing(Turing)` should not have halted. In the second case, if `Turing(Turing)` does not halt, then it must be the case that `TestHalt(Turing, Turing)` returned "yes," which would mean that `Turing(Turing)` should have halted. In both cases, we arrive at a contradiction which must mean that our initial assumption, namely that the program `TestHalt` exists, was wrong. Thus, `TestHalt` cannot exist, so it is impossible for a program to definitively check if any general program halts! □

What proof technique did we use? This was actually a proof by diagonalization, the same technique that we used earlier to show that the real numbers are uncountable! Why? Since the set of all computer programs is countable (they are, after all, just finite-length strings over some alphabet, and the set of all finite-length strings is countable), we can enumerate all programs as follows (where P_i represents the i^{th} program):



	P0	P1	P2	P3	P4 ...
P0	Ⓗ	L	H	L	H ...
P1	L	Ⓛ	H	L	L ...
P2	H	H	Ⓗ	H	L ...
P3	H	H	H	Ⓗ	H ...
⋮					

The $(i,j)^{\text{th}}$ entry in the table above is H if program P_i halts on input P_j, and L (for "Loops") if it does not halt. (Here, we assume that malformed programs with syntax errors just halt with an error.) Now if the program `Turing` exists it must occur somewhere on our list of programs, say as P_n. But this cannot be, since if the n^{th} entry in the diagonal is H, meaning that P_n halts on P_n, then by its definition `Turing` loops on P_n; and if the entry is L, then by definition `Turing` halts on P_n. Thus the behavior of `Turing` is different from that of P_n, and hence `Turing` does not appear on our list. Since the list contains all possible programs, we must conclude that the program `Turing` does not exist. And since `Turing` is constructed by a simple modification of `TestHalt`, we can conclude that `TestHalt` does not exist either. Hence the Halting Problem cannot be solved.

In fact, there are many more questions we would like to answer about programs but cannot answer decisively. For example, we cannot definitively know if a program ever outputs anything or if it ever executes a specific line. We also cannot definitively check if two programs produce the same output. And we cannot definitively check to see if a given program is a virus. To attempt to do any of these tasks, we have to allow our approach to answer "I don't know" and give up. Otherwise, the halting problem's impossiblity would infect our approach as well. These issues are explored in greater detail in the advanced course CS172 (Computability and Complexity).

Uncomputable numbers

The fact that the real numbers are uncountably infinite and that there are only a countable number of computer programs tells us that the vast majority of real numbers are fundamentally unknowable to computers. The halting problem above tells us that many of them are also interesting. For example, consider the real number between 0 and 1 whose ith binary digit is 0 if the ith computer program doesn't halt and is 1 if the ith computer program does halt. The halting problem argument tells us that this number is uncomputable.

In a very related proof, the logician Gödel showed that the following real number is also uncomputable. Consider all finite strings of mathematical symbols involving $\forall, \exists$, variables, as well as the arithmetic operations $+, *, -, /$ and exponentiation, comparisons $=, <, >$ and the logical operators $\neg, \wedge, \vee, \implies$. A string like that is either a syntax error or it is a valid proposition about the natural numbers. All such finite strings are certainly countable. So we can talk about the ith such string. Consider the real number between 0 and 1 in which the ith digit is 0 if the string is a syntax error or the proposition it represents is false. The ith digit is 1 if the string is well-formed and the proposition it represents is true (i.e. there is no counterexample to it). The resulting real number is uncomputable.

This turns out to mean that there are true statements about the integers for which there is no proof. In a very real sense, they just happen to be true for no good[4] reason. The even more surprising consequence is that the same uncomputability result holds if we replace "true" and "false" with "provable" and "unprovable." Here, we can consider a proposition provable if it is either a syntax error (syntax can be checked by a program), has a counterexample, or has a valid proof. It is unprovable otherwise. So, not only are there lots of unprovable assertions out there (if there were a finite number, we could simply have a finite list of them that a computer program could check) but they are impossible for a computer program to reliably recognize[5] as unprovable.

The proof of these facts is beyond the scope of the course, but is related to the deep connection between

[4] To be more precise, they are true because they are true in the specific concrete model you have for the integers.

[5] Being a valid proof can be checked by a computer program. Each statement has to follow logically from those that came before. The number of potential proofs is countably infinite. The problem is that the program that simply compares statements to all valid proofs is not guaranteed to halt! Because there are unprovable statements, it might just run forever and never encounter neither a proof nor a counterexample.

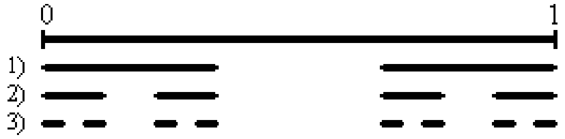
proofs and computation. Computation is a kind of living proof.

Advanced Material On Counting and Computability

Please read on only if interested.

The Cantor Set

The Cantor set is a remarkable set construction involving the real numbers in the interval $[0,1]$. The set is defined by repeatedly removing the middle thirds of line segments infinitely many times, starting with the original interval. For example, the first iteration would involve the removal of the (open) interval $(\frac{1}{3}, \frac{2}{3})$, leaving $[0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$. We then proceed to remove the middle third of each of these two remaining intervals, and so on. The first three iterations are illustrated below:



The Cantor set contains all points that have *not* been removed: $C = \{x : x \text{ not removed}\}$. How much of the original unit interval is left after this process is repeated infinitely? Well, we start with an interval of length 1, and after the first iteration we remove $\frac{1}{3}$ of it, leaving us with $\frac{2}{3}$. For the second iteration, we keep $\frac{2}{3} \times \frac{2}{3}$ of the original interval. As we repeat these iterations infinitely often, we are left with:

$$1 \longrightarrow \tfrac{2}{3} \longrightarrow \tfrac{2}{3} \times \tfrac{2}{3} \longrightarrow \tfrac{2}{3} \times \tfrac{2}{3} \times \tfrac{2}{3} \longrightarrow \cdots \longrightarrow \lim_{n\to\infty} (\tfrac{2}{3})^n = 0$$

According to this calculation, we have removed everything from the original interval! Does this mean that the Cantor set is empty? No, it doesn't. What it means is that the *measure* of the Cantor set is zero; the Cantor set consists of isolated points and does not contain any non-trivial intervals. In fact, not only is the Cantor set non-empty, it is uncountable![6]

To see why, let us first make a few observations about ternary strings. In ternary notation, all strings consist of digits (called "trits") from the set $\{0,1,2\}$. All real numbers in the interval $[0,1]$ can be written in ternary notation. (E.g., $\frac{1}{3}$ can be written as 0.1, or equivalently as $0.0222\ldots$, and $\frac{2}{3}$ can be written as 0.2 or as $0.1222\ldots$.) Thus, in the first iteration, the middle third removed contains all ternary numbers of the form 0.1xxxxx. The ternary numbers left after the first removal can all be expressed either in the form 0.0xxxxx... or 0.2xxxxx... (We have to be a little careful here with the endpoints of the intervals; but we can handle them by writing $\frac{1}{3}$ as $0.02222\ldots$ and $\frac{2}{3}$ as 0.2.) The second iteration removes ternary numbers of the form 0.01xxxxx and 0.21xxxxx (i.e., any number with 1 in the second position). The third iteration removes 1's in the third position, and so on. Therefore, what remains is all ternary numbers consists of only 0's and 2's. Thus we have shown that

$$C = \{x \in [0,1] : x \text{ has a ternary representation consisting only of 0's and 2's}\}.$$

[6] It's actually easy to see that C contains at least countably many points, namely the endpoints of the intervals in the construction—i.e., numbers such as $\frac{1}{3}$, $\frac{2}{3}$, $\frac{1}{9}$, $\frac{1}{27}$ etc. It's less obvious that C also contains various other points, such as $\frac{1}{4}$ and $\frac{3}{10}$. (Why?)

Finally, using this characterization, we can set up an *onto* map f from C to $[0,1]$. Since we already know that $[0,1]$ is uncountable, this implies that C is uncountable also. The map f is defined as follows: for $x \in C$, $f(x)$ is defined as the binary decimal obtained by dividing each digit of the ternary representation of x by 2. Thus, for example, if $x = 0.0220$ (in ternary), then $f(x)$ is the binary decimal 0.0110. But the set of all binary decimals 0.xxxxx... is in 1-1 correspondence with the real interval $[0,1]$, and the map f is onto because every binary decimal is the image of some ternary string under f (obtained by doubling every binary digit).[7] This completes the proof that C is uncountable.

Power Sets and Higher Orders of Infinity

Let S be any set. Then the *power set* of S, denoted by $\mathscr{P}(S)$, is the set of all subsets of S. More formally, it is defined as: $\mathscr{P}(S) = \{T : T \subseteq S\}$. For example, if $S = \{1,2,3\}$, then $\mathscr{P}(S) = \{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$.

What is the cardinality of $\mathscr{P}(S)$? If $|S| = k$ is finite, then $|\mathscr{P}(S)| = 2^k$. To see this, let us think of each subset of S corresponding to a k-bit string, where a 1 in the ith position indicates that the ith element of S is in the subset, and a 0 indicates that it is not. In the example above, the subset $\{1,3\}$ corresponds to the string 101. Now the number of binary strings of length k is 2^k, since there are two choices for each bit position. Thus $|\mathscr{P}(S)| = 2^k$. So for finite sets S, the cardinality of the power set of S is exponentially larger than the cardinality of S. What about infinite (countable) sets? We claim that there is no bijection from S to $\mathscr{P}(S)$, so $\mathscr{P}(S)$ is not countable. Thus for example the set of all subsets of natural numbers is not countable, even though the set of natural numbers itself is countable!

Theorem: $|\mathscr{P}(\mathbb{N})| > |\mathbb{N}|$.

Proof: Suppose towards a contradiction that there is a bijection $f : \mathbb{N} \to \mathscr{P}(\mathbb{N})$. Recall that we can represent a subset by a binary string, with one bit for each element of $\mathbb{N}$. (So, since $\mathbb{N}$ is infinite, the string will be infinitely long. Contrast the case of $\{0,1\}^*$ discussed earlier, which consists of all binary strings of *finite* length.) Consider the following diagonalization picture in which the function f maps natural numbers x to binary strings which correspond to subsets of $\mathbb{N}$:

[7] Note that f is *not* injective; for example, the ternary strings $0.20222\ldots$ and 0.22 map to binary strings $0.10111\ldots$ and 0.11 respectively, which denote the same real number. Thus f is not a bijection. However, the current proof shows that the cardinality of C is at least that of $[0,1]$, while it is obvious that the cardinality of C is at most that of $[0,1]$ since $C \subset [0,1]$. Hence by Cantor-Schröder-Bernstein C has the same cardinality as $[0,1]$ (and as $\mathbb{R}$).

$$\begin{array}{c|ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\ \hline 0 & \textcircled{1} & 0 & 0 & 0 & 0 & 0 & \dots \\ 1 & 0 & \textcircled{0} & 0 & 0 & 0 & 0 & \dots \\ 2 & 1 & 0 & \textcircled{1} & 0 & 0 & 0 & \dots \\ 3 & & & & & & & \\ \vdots & & & & & & & \end{array}$$

In this example, we have assigned the following mapping: $0 \rightarrow \{0\}$, $1 \rightarrow \{\}$, $2 \rightarrow \{0{,}2\}$, ... (i.e., the nth row describes the n^{th} subset as follows: if there is a 1 in the k^{th} column, then k is in this subset, else it is not.) Using a similar diagonalization argument to the earlier one, flip each bit along the diagonal: $1 \rightarrow 0$, $0 \rightarrow 1$, and let b denote the resulting binary string. First, we must show that the new element is a subset of $\mathbb{N}$. Clearly it is, since b is an infinite binary string which, as we've seen, corresponds to a subset of $\mathbb{N}$. Now suppose b were the n^{th} set in the list. This cannot be the case though, since the n^{th} bit of b differs from the n^{th} bit of the diagonal (the bits are flipped). So it's not on our list, but it should be, since we assumed that the list enumerated all possible subsets of $\mathbb{N}$. Thus we have a contradiction, implying that $\mathscr{P}(\mathbb{N})$ is uncountable.

Thus we have seen that the cardinality of $\mathscr{P}(\mathbb{N})$ (the power set of the natural numbers) is strictly larger than the cardinality of $\mathbb{N}$ itself. The cardinality of $\mathbb{N}$ is denoted $\aleph_0$ (pronounced "aleph null"), while that of $\mathscr{P}(\mathbb{N})$ is denoted $2^{\aleph_0}$. It turns out that in fact $\mathscr{P}(\mathbb{N})$ has the same cardinality as $\mathbb{R}$ (the real numbers), and indeed as the real numbers in $[0,1]$. This cardinality is known as $\mathbf{c}$, the "cardinality of the continuum." So we know that $2^{\aleph_0} = \mathbf{c} > \aleph_0$. Even larger infinite cardinalities (or "orders of infinity"), denoted $\aleph_1, \aleph_2, \ldots$, can be defined using the machinery of set theory; these obey (to the uninitiated somewhat bizarre) rules of arithmetic. Several fundamental questions in modern mathematics concern these objects. For example, the famous "continuum hypothesis" asserts that $\mathbf{c} = \aleph_1$ (which is equivalent to saying that there are no sets with cardinality between that of the natural numbers and that of the real numbers).

2.2 Self-Replicating Programs

Can we use self-reference to design a program that outputs itself? To illustrate the idea, let us consider how we can do this if we could write the program in English. Consider the following instruction:

```
Print out the following sentence:

    "Print out the following sentence:"
```

If we execute the instruction above (interpreting it as a program), then we will get the following output:

```
    Print out the following sentence:
```

Clearly this is not the same as the original instruction above, which consists of two lines. We can try to modify the instruction as follows:

```
Print out the following sentence twice:

    "Print out the following sentence twice:"
```

Executing this modified instruction yields the output which now consists of two lines:

```
Print out the following sentence twice:
```

```
Print out the following sentence twice:
```

This almost works, except that we are missing the quotes in the second line. We can fix it by modifying the instruction as follows:

```
Print out the following sentence twice, the second time in quotes:

    “Print out the following sentence twice, the second time in quotes:”
```

Then we see that when we execute this instruction, we get exactly the same output as the instruction itself:

```
Print out the following sentence twice, the second time in quotes:

    “Print out the following sentence twice, the second time in quotes:”
```

2.2.1 Quines and the Recursion Theorem

In the above section we have seen how to write a self-replicating program in English. But can we do this in a real programming language? In general, a program that prints itself is called a *quine*,[8] and it turns out we can always write quines in any programming language.

As another example, consider the following pseudocode:

```
(Quine “s”)

    (s “s”)
```

The pseudocode above defines a program `Quine` that takes a string `s` as input, and outputs `(s “s”)`, which means we run the string `s` (now interpreted as a program) on itself. Now consider executing the program `Quine` with input `“Quine”`:

```
(Quine “Quine”)
```

By definition, this will output

```
(Quine “Quine”)
```

which is the same as the instruction that we executed!

This is a simple example, but how do we construct quines in general? The answer is given by the *recursion theorem*. The recursion theorem states that given any program $P(x,y)$, we can always convert it to another program $Q(x)$ such that $Q(x) = P(x,Q)$, i.e., Q behaves exactly as P would if its second input is the description of the program Q. In this sense we can think of Q as a “self-aware” version of P, since Q essentially has access to its own description.

[8] Quine is named after the philosopher and logician Willard Van Orman Quine, as popularized in the book *“Gödel, Escher, Bach: An Eternal Golden Braid”* by Douglas Hofstadter.

The Easy Halting Problem

As noted above, the key idea in establishing the uncomputability of the Halting Problem is self-reference: Given a program P, we ran into trouble when deciding whether $P(P)$ halts. But in practice, how often do we want to execute a program with its own description as input? Is it possible that if we disallow this kind of self-reference, we can solve the Halting Problem?

For example, given a program P, what if we ask instead the question: "Does P halt on input 0?" This looks easier than the Halting Problem (hence the name "Easy Halting Problem"), since we only need to check whether P halts on a specific input 0, instead of an arbitrary given input (such as P itself). However, it turns out that this seemingly easier problem is still uncomputable! We prove this claim by showing that if we could solve the Easy Halting Problem, then we could also solve the Halting Problem itself; since we already know that the Halting Problem is uncomputable, this implies that the Easy Halting Problem must also be uncomputable.

Specifically, suppose we have a program `TestEasyHalt` that solves the Easy Halting Problem:

$$\texttt{TestEasyHalt(P)} = \begin{cases} \text{“yes",} & \text{if program } P \text{ halts on input } 0 \\ \text{“no",} & \text{if program } P \text{ loops on input } 0 \end{cases}$$

Then we can use `TestEasyHalt` as a subroutine in the following program that solves the Halting Problem:

```
Halt(P,x)
  construct a program P′ that, on input 0, returns P(x)
  return TestEasyHalt(P′)
```

The algorithm `Halt` constructs another program P', which depends on both the original program P and the original input x, such that the call $P'(0)$ returns $P(x)$. Such a program P' can be constructed very simply as follows:

```
P′(y)
  return P(x)
```

That is, the new program P' ignores its input y and always returns $P(x)$. (Note that the descriptions of P and of x are "hard-wired" into P'.) Then we see that $P'(0)$ halts if and only if $P(x)$ halts. Therefore, if we have such a program `TestEasyHalt`, then `Halt` will correctly solve the Halting Problem. Since we know there cannot be such a program `Halt`, we conclude `TestEasyHalt` does not exist either.

The technique that we use here is called a *reduction*. Here we are reducing one problem "Does P halt on x?" to another problem "Does P' halt on 0?", in the sense that if we know how to solve the second problem, then we can use that knowledge to construct an answer for the first problem. This implies that the second problem is actually as difficult as the first, despite the apparently simpler description of the second problem. Reductions are a powerful tool both for proving impossibility results such as these, and for designing algorithms for new problems from existing algorithms. Both of these directions are explored further in CS172 and CS170.

2.3 Godel's Incompleteness Theorem

In 1900, the great mathematician David Hilbert posed the following two questions about the foundation of logic in mathematics:

1. Is arithmetic consistent?
2. Is arithmetic complete?

To understand the questions above, we recall that mathematics is a formal system based on a list of axioms (for example, Peano's axioms of the natural numbers, axiom of choice, etc.) together with rules of inference. The axioms provide the initial list of true statements in our system, and we can apply the rules of inference to prove other true statements, which we can again use to prove other statements, and so on.

The first question above asks whether it is possible to prove both a proposition P and its negation $\neg P$. If this is the case, then we say that arithmetic is *inconsistent*; otherwise, we say arithmetic is *consistent*. If arithmetic is inconsistent, meaning there are false statements that can be proved, then the entire arithmetic system will collapse because from a false statement we can deduce anything, so every statement in our system will be vacuously true.

The second question above asks whether every true statement in arithmetic can be proved. If this is the case, then we say that arithmetic is *complete*. We note that given a statement, which is either true or false, it can be very difficult to prove which one it is. As a real-world example, consider the following statement, which is known as Fermat's Last Theorem:

$$(\forall n \geq 3)\ \neg(\exists x, y, z \in \mathbb{Z}^+)(x^n + y^n = z^n).$$

This theorem was first stated by Pierre de Fermat in 1637,[9] but it had eluded proofs for centuries until it was finally proved by Andrew Wiles in 1994.

In 1928, Hilbert formally posed the questions above as the *Entscheidungsproblem*. Most people believed that the answer would be "yes," since ideally arithmetic should be both consistent and complete. However, in 1930 Kurt Gödel proved that the answer is in fact "no": Any formal system that is sufficiently rich to formalize arithmetic is either inconsistent (there are false statements that can be proved) or incomplete (there are true statements that cannot be proved). Gödel proved his result by exploiting the deep connection between proofs and arithmetic. Actually Gödel's theorem also embodies a deep connection between proofs and computation, which was illuminated after Turing formalized the definition of computation in 1936 via the notion of Turing machines and computability.

In the rest of this note, we will first sketch the essence of Gödel's proof, and then outline an easier proof of the theorem using what we know about the Halting Problem.

2.3.1 Sketch of Gödel's Proof

Suppose we have a formal system F, which consists of a list of axioms and rules of inference, and assume F is sufficiently expressive that we can use it to express all of arithmetic.

[9] Along with the famous note: "I have discovered a truly marvelous proof of this, which this margin is too narrow to contain."

Now suppose we can write the following statement in arithmetic:

$$S = \text{“This statement is not provable in } F\text{.”}$$

Once we have this statement, there are two possibilities:

1. Case 1: S is provable. Then the statement S is true, but by inspecting the content of the statement itself, we see that this implies S should not be provable. Thus, F is inconsistent in this case.
2. Case 2: S is not provable. By construction, this means the statement S is true. Thus, F is incomplete in this case, since there is a true statement (namely, S) that is not provable.

To complete the proof, it now suffices to construct such a statement S. This is the difficult part of Gödel's proof, which requires a clever encoding (a so-called "Gödel numbering") of symbols and propositions as natural numbers.

2.3.2 Proof via the Halting Problem

Let us now see how we can prove Gödel's result by reduction to the Halting Problem. Here we proceed by contradiction: Suppose arithmetic is both consistent and complete; we will use this assumption to solve the Halting Problem, which we have seen is impossible.

Recall that in the Halting Problem we want to decide whether a given program P halts on a given input x. For fixed P and x, let $S_{P,x}$ denote the proposition "P halts on input x." The key observation is that this proposition can be phrased as a statement in arithmetic. The form of the statement $S_{P,x}$ will be

$$\exists z (z \text{ encodes a valid halting execution sequence of } P \text{ on input } x).$$

Although the details require some work, your programming intuition should hopefully convince you that such a statement can be written, in a fairly mechanical way, using only the language of standard arithmetic, with the usual operators, connectives and quantifiers: basically the statement just has to check, step by step, that the string z (encoded as a very long integer in binary) lists out the sequence of states that a computer would go through when running program P on input x.

Now let us assume, for contradiction, that arithmetic is both consistent and complete. This means that, for any (P,x), the statement $S_{P,x}$ is either true or false, and that there must exist a proof in arithmetic of either $S_{P,x}$ or its negation, $\neg S_{P,x}$ (and not both). But now recall that a proof is simply a finite binary string. Therefore, there are only countably many possible proofs, so we can enumerate them one by one and search for a proof of $S_{P,x}$ or $\neg S_{P,x}$. The following program performs this task:

```
Search(P,x)
  for every proof q:
    if q is a proof of $S_{P,x}$ then output "yes"
    if q is a proof of $\neg S_{P,x}$ then output "no"
```

The program Search takes as input the program P, and proceeds to check every possible proof until it finds either one that proves $S_{P,x}$, or one that proves $\neg S_{P,x}$. By assumption, we know that one of these proofs always exists, so the program Search will terminate in finite time, and it will correctly solve the Halting Problem. On the other hand, since we have already established that the Halting Problem is uncomputable, such a program Search cannot exist. Therefore, our initial assumption must be wrong, so it is not true that arithmetic is both consistent and complete.

Note that in the argument above we rely on the fact that, given a proof, we can construct a program that mechanistically checks whether it is a valid proof of a given proposition. This is a manifestation of the intimate connection between proofs and computation.