Computer Science

Dr. Y. Liow (May 7, 2024)

Contents

Chapter 15

Turing Machines

debug: chap.t

15.1 Turing Machines debug: tm.tex

Informally, a Turing machine (TM) is like a DFA or a PDA except for the following:

- For the DFA (and PDA), the machines read the input string one character at a time from the leftmost character to the rightmost. A TM can also read the input one character at a time. However the input string should be thought of as being written on an infinitely long tape; you should think of the tape as being padded infinitely to the right with blanks. The TM can actually write to it as well. Therefore you should think of the TM as having a read/write head that moves along an infinitely long string. Furthermore, the read/write head can move either left or right or it can also stay. The only restriction is that it must stay on the infinitely tape.
- For the DFA, acceptance is determined by what the state the machine is in once the string is completely read. For the TM, since the input string is written on an infinite tape, there is really no end-of-string. Therefore for a TM, a string is accepted if the machine enters an accepting state, regardless of the position of the read/write head.

Even without a formal definition of a TM, you probably know by now that the most important thing about running the TM is that you need to know what happens when the machine is in a particular state and it is about to read a particular character. What if the machine capable of doing? Well, obviously it has to go to another (possibly the same) state. It can overwrite the character it is reading; this is sort of like the PDA. Furthermore it can move it's read/write head either to the left or right or it stays. So the transition function should look like:

$$\delta(q, a) = (p, b, D)$$

where q and p are states, a and b are the allowable characters on the infinite tape and D refers to the direction moved by the read/write head. D can take values L (for left), S (for stay) and R (for right).

Here's the formal definition. Most of this won't be a surprise to you by now. There's one thing I should mention. We will distinguish between the alphabet of the string used to generate a language (the Σ) and the alphabet that the TM can write on the tape.

Definition 15.1.1. A **Turing machine** TM M is made up of

- Q: a finite set of states.
- Σ: a finite set of input alphabet.
- Γ: a finite set of alphabet the TM using for read and write. Σ is a subset of Γ. Furthermore the blank character is also in Γ.
- q_0: the initial state. This is in Q.
- F: the set of accepting states. F is a subset of Q.
- R: the set of reject states. R is a subset of Q.
- B: the blank symbol; this is in Γ but not in Σ. In most TMs (and textbooks), this character is standardized and the same, so it's usually not specified when describing a TM.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$ is the transition function.

There are many minor variations of the TM definition and they are all essentially the same. For instance:

1. Instead of having a set of accept states F, it's possible to redefine the TM to have one single accept state without changing what the TM does. In this case, the single accept state is usually written q_{accept}.
2. Likewise it's possible to redefine a TM to have only one reject state. In this case the special reject state is written q_{reject}.
3. It's possible to rewrite the TM so that the read/write head does not stay but always either move left or right. In this case, the transition function is of the form:
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

Here's how you draw a TM given the above specification: Everything is the same as a DFA. For the transitions, suppose you have

$$\delta(q, a) = (p, b, R)$$

Then you draw a directed edge or transition from state q to state p and label it $a/b, R$ or $a \to b, R$

To be formal about computation we will introduce the following notation:

Definition 15.1.2. Let

$$x_1 \dots x_m q x_{m+1} \dots x_n$$

to denote that fact that the TM current has it's read/write head pointing to

the $(m+1)$-st character on the tape x_{m+1}; this will be the *next* character the TM will read. This is called an **instantaneous description** (ID). In some books this is also called a **configuration**. Of course if you're running the TM with string x, the initial ID is

$$q_0 x$$

Of course these are all just notation for description a computation of the TM. In particular if $\delta(q, a) = (b, p, R)$, then

$$xqay \vdash xbpy$$

The read/write head over-writes a with b and moves right. Make sure you understand that! If instead of that you have $\delta(q, a) = (b, p, L)$, then

$$xcqay \vdash xpcby$$

so that the character being read, a, is over-written by b and the read/write head moves left. And if the transition if $\delta(q, a) = (b, p, S)$, then the read/write head stays:

$$xqay \vdash xpby$$

As always, if we will use $\vdash^*$ to note "the same or at least one derivation.

Definition 15.1.3. $x \in \Sigma^*$ is **accepted** by M is there is some $p \in F$ such that

$$q_0 x \vdash^* ypz$$

for some strings y, z in Γ^*. And, surprise-surprise, $L(M)$ is the set of strings accepted by M.

Definition 15.1.4.

- It's easy to show that in fact you only need one accepting state. (Why?) Therefore some books will just have a special state called q_{accept} for the only accepting state of the TM.
- Note that if the TM tries to move to the left while it's already at the leftmost position on the tape, then the machine crashes and stops running. The string is not accepted. Instead of allowing this to happen, some books will define a special state q_{reject}. Whenever the TM enters the q_{reject} state it stops (or halt) and the string is not accepted.
- From the above, the TM can reject by either entering the state q_{reject} or by moving left on the leftmost position on the tape. It's easy to see that you really do not need both cases. You can rewrite the TM so that if it tries to fall off the left edge, you make the TM go into q_{reject} instead. Here's how you do it. Before running the TM, shift all the checks of the

input string to the right by one, and insert a "beginning of tape marker"; you can use any symbol not used for instance a common symbol in some books is '$'. Then include a transition so that is the TM is reading the "beginning of tape marker" and it attempts to move the left, replace it with a transition that enters q_{reject} instead. This will stop the machine before it falls off the left edge.

- Although according to the above definition of a TM there is a transition for every state q and every symbol a in Γ, it is customary not to include transitions that enters the state q_{reject}. Therefore in some books some transitions are left out and the authors say that "if the TM is in a state q and is reading a character a but there is no applicable transition, the machine crashes and the string is rejected." This is just the same as including a transition from such a state and symbol to the rejecting state and of course you cannot exit the rejecting state.

Definition 15.1.5. A language is said to be **Turing recognizable** or **recursively enumerable** (r.e.) if it is accepted by a TM.

(Note: Computer Science is so new that many concepts still have multiple names. Tune in after 100 hundred years to find out which name finally gets picked.)

Note one curious feature of the TM. It is possible for it to run forever. That's not the case for either the DFA, NFA or PDA. In particular it's possible for the string not to accepted by the TM entering the q_{reject} state, the TM crashes by moving to the left while at the leftmost position on the tape, or it does not enter an accepting state.

Definition 15.1.6. A language is said to be **recursive** (rec.) if it is accepted by a TM that always halts. A TM **always halts** if giving any input string, the TM will either reach an accepting state or the rejecting state. The accepting state and rejecting state are called **halting states**.

Example 15.1.1. Let $L = \{a^n b^n c^n \mid n \geq 0\}$. We know that L is not a CFL. We will now prove that it is recursively enumerable, i.e., accepted by a TM.

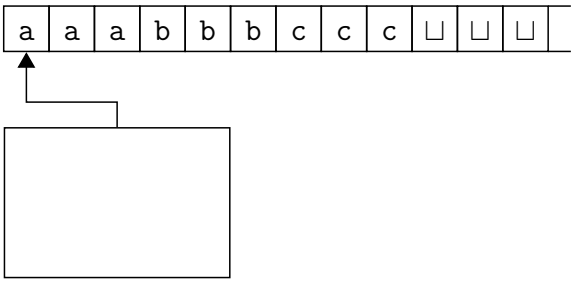
TMs tend to large. So to simplify the specification of this TM, transitions entering the q_{reject} state is not listed.

The idea is simple: Scan left and right, marking a single a and a single b and a single c in each right scan. We will need special characters to denote that

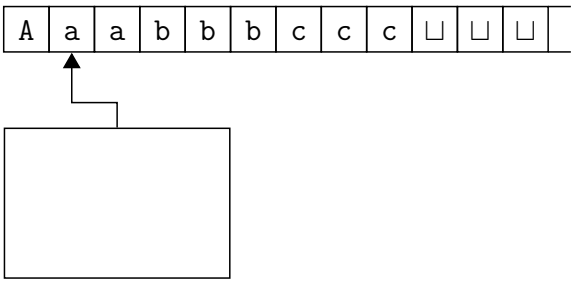
a,b and c are marked. We will use A, B and C respectively. Of course if we cannot find a b or c to match this a, the string is rejected. Now once c is marked, we move all the way to the left to look for the first marked a (i.e., A). Once that's found, we move right. This would either be another a or B. If it's a, we repeat the process of marking a,b,c.

On the other hand if it's B, then there are no more a's. Then we just move right over all the marked characters (i.e., B and C) until we see a blank and accept the string. Of course if b or c is found, then the string is rejected; there shouldn't be anymore b's or c's for the string to be accepted.

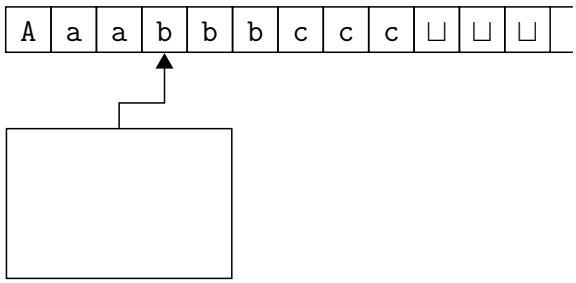
That's the main idea. Picturially, this is what the TM should do. Initially we have this, where the input is *aaabbbccc*. We will use $\sqcup$ to denote the blank character.



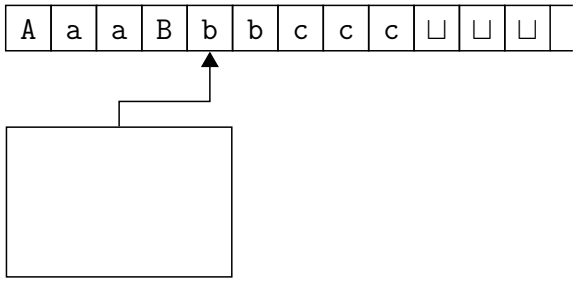
At this state (the initial state), it reads and a and replace a by A and move right (the read/write head) by one step:



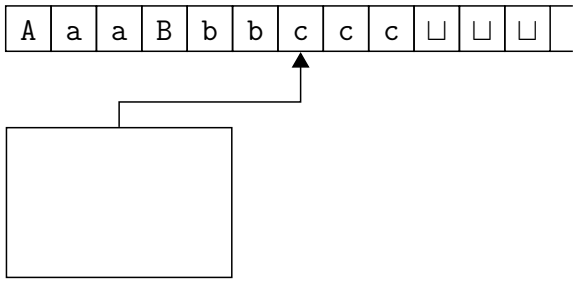
Again an A means "checked a". Now at this point (meaning at this state), the TM just keep moving right whenever it sees an a (without changing what is read) until it reads a b:



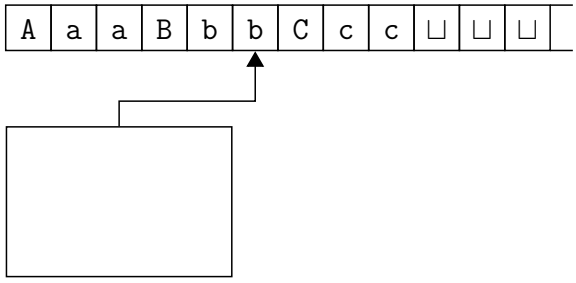
At this point (meaning at this state), the TM replaces the b with B (in other words, mark the b as read), and move right by one step:



At this point, the TM moves to the right as long as it reads a b. The TM will then reach this state:



The TM then reads c, replaces it with a C and moves left by one.



The TM has nw finished one round of match one a, one b, and one c.

(q, x)	$\delta(q, x)$	
(q_0, a)	(q_1, A, R)	Start marking phase: Mark a and move right
(q_1, a)	(q_1, a, R)	Fast forward over a
(q_1, B)	(q_1, B, R)	Fast forward over B
(q_1, b)	(q_2, B, R)	Mark b and move right
(q_2, b)	(q_2, b, R)	Fast forward over b
(q_2, C)	(q_2, C, R)	Fast forward over C
(q_2, c)	(q_3, C, L)	Mark c and move left
(q_3, C)	(q_3, C, L)	Rewind over C
(q_3, B)	(q_3, B, L)	Rewind over B
(q_3, b)	(q_3, b, L)	Rewind over b
(q_3, a)	(q_3, a, L)	Rewind over a
(q_3, A)	(q_0, A, R)	On seeing A, go to state q_0
(q_0, B)	(q_4, B, R)	Start scanning phase: read first B and move right
(q_4, B)	(q_4, B, R)	Fast forward over B
(q_4, C)	(q_5, C, R)	Read first C and move right
(q_5, C)	(q_5, C, R)	Fast forward over C
$(q_5, \sqcup)$	$(q_{\text{accept}}, B, S)$	Accept string when a $\sqcup$ is found

Make sure you draw the state (or transition) diagram. Again transitions which are not labeled go to the q_{reject} state. Instead of writing infinitely many blanks, we will write one blank just beyond abc. If we need to we can add blanks when we need to (actually, for this TM only one blank is needed.)

Now let's trace the execution of M for the string abc.

$$\begin{aligned} q_0abc\sqcup &\vdash Aq_1bc\sqcup \\ &\vdash ABq_2c\sqcup \\ &\vdash Aq_3BC\sqcup \\ &\vdash q_3ABC\sqcup \\ &\vdash Aq_0BC\sqcup \\ &\vdash ABq_4C\sqcup \\ &\vdash ABCq_5\sqcup \\ &\vdash ABCq_{\text{accept}}\sqcup \end{aligned}$$

Now try to trace the execution of the TM for $aabbcc$ and $aabbccc$.

Example 15.1.2. You can use TM to compute numbers. For instance you can use the string 111 to represent 3. Given this data format, you can define the output of a TM M by what's on the tape when then machine halts. Can you define a TM that computes the twice function. In other words $M(111) = 111111$, i.e., if you put 111 on the tape and run the machine, it should halt with 111111 on the tape. If the TM does not halt on input x, then we say $M(x)$ **diverges**.

- You can design a TM to add numbers. For instance to add two 3 and 5, the input to the TM is #111#11111. The expected output is 11111111. Can you design such a TM?
- Can you design one to perform subtract of the form $m-n$ where $m \geq n$? In otherwise you want to design a TM M such that $M(\#11111\#11) = 111$.
- Can you design a TM to do multiplication? For instance to compute the product of 3 and 4, you run the machine with 111#1111 and get $M(111\#1111) = 111111111111$.
- Can you design a TM M that can powers of 2? For instance $M(1) = 11$, $M(111) = 11111111$.
- Of course you can also compute rationals. For instance you can represent the fraction 3/4 by 11101111. Can you design a TM to add rationals?
- You can also model negative numbers just by using a special character for sign.
- Of course you can combine all the above into a single TM.

Example 15.1.3. Here's a trick. To prevent the case where the TM moves left while it's already at the leftmost position, you need to shift all the character one position to the right. Suppose the symbol # is not used. Suppose the TM is M. How would you write a new TM that will shift the input by one space to the right, put # as the first character, and move the read/write head so that it's about to read the first character of the input string (i.e., first character to the right of #. Once this pre-processing step is done, the new TM starts simulating the old one. In other words in terms of ID, you want a TM M' that will do the following: Suppose the new start state is q_0'.

$$q_0 x \vdash \# q_0 x$$

Since this modification can always be carried out on any TM, in many books, it's assumed that the TM will never fall off the left edge. In other words, these authors assume that it the TM tries to move left on the leftmost spot, it will simply stay and not move.

Example 15.1.4. Design a TM that accepts $\{a^n b^n c^n \mid n \equiv 1 \pmod 4\}$. The solution is easy. First the TM checks the input is of the form $a^n b^n c^n$. (We've already done that). After this point if the a's might be marked with another character. Let's say it's been replaced by A's. First the TM go to the leftmost A. It moves right so that the read/write head is reading the next A (or possibly B). It will then continually read 4 A's until a B is reached. If that can be done, then TM enters the q_{accept} state. Otherwise it enters the q_{reject} state.

There are many other computational models (λ–calculus random access machines, general recursive functions, etc.) But in the end they are all proven to be only as powerful as the original Turing machines. Therefore solving a problem algorithmically is believed to be equivalent to solving a problem using a Turing machine. This is known as the **Church-Turing thesis**.

Example 15.1.5. In some books, a Turing machine is defined very similar to our definition above except that the movements of the read/write head is either left or right; the read/write head cannot stay at the same spot. It seems that such a definition would result in a weaker TM. Is that true?

15.2 Back to Basics ... Counting (a la Cantor)

debug: counting.tex

Let's being by reviewing what we mean by counting. When I give you the following:

A, B, C, D, E

you should tell me that you are seeing 5 things. Why is that? Because since you started to learn, you were told to associate 1, 2, 3, ... to the things you see. So for instance in the above, you would do put your finger on A and say "one", move your finger to B and say "two", etc. That's the same as doing this:

A	B	C	D	E
↑	↑	↑	↑	↑
1	2	3	4	5

i.e., associating the number 1 to A (instead of associating your verbal "one" to A), etc. This association (think "function") is 1–1 and onto (if you were taught to count correctly!) If it's not 1–1, it might be something like this: you might have for instance 1 and 2 pointing to A, you would be counting A twice. If the association is not onto, you would be missing some letter.

Another way to think of counting the above is there you're finding a 1–1 and onto function from the set

$$\{1, 2, 3, 4, 5\}$$

to the set

$$\{\text{A, B, C, D, E}\}$$

Therefore counting

A, B, C, D, E

is the same as trying to a 1–1 and onto function from

$$\{1\}$$

to $\{A, B, C, D, E\}$, or from

$$\{1, 2\}$$

to $\{A, B, C, D, E\}$, or from

$$\{1, 2, 3\}$$

etc. until you succeed. In our case we succeed when we use

$$\{1, 2, 3, 4, 5\}$$

You can think of

$$\{1\},\ \{1, 2\},\ \{1, 2, 3\},\ \{1, 2, 3, 4\},\ \{1, 2, 3, 4, 5\},\ \{1, 2, 3, 4, 5, 6\},\ \{1, 2, 3, 4, 5, 6, 7\}, ...$$

as our standard measurements for counting sort of like standard weights for weighing things. But why should we use 1–1 and onto functions and these sets as a way to define counting?!? Counting is so basic and such a primitive concept, why bother trying to view counting as something involving sets, functions, and 1–1 and onto functions?!? Shouldn't "counting" be simpler than "1–1, onto functions"???

Well, the reason is because this definition of counting uses sets ... and sets can be infinite. For instance I can use the whole set of natural numbers to count the number of things in the set X:

$$\{0, 1, 2, 3, ...\} \to X$$

and the standard measuring set $\{0, 1, 2, 3, ...\}$ is not finite!!! (Remember: depending on who you talk to the set $\mathbb{N}$ might or might not contain 0.)

Therefore if you want to say that two sets X and Y are different, one way would be to say that X has as many things as $\mathbb{N} = \{0, 1, 2, 3, ...\}$ while Y has as many things as $\mathbb{R}$ ... that is if you know that $\mathbb{R}$ is "bigger" than $\mathbb{N}$ which I will show you in a minute.

First of all let me say that two sets X and Y are equinumerous (or informally, they have the same size) if there is a 1–1 and onto function from X to Y. In that case I will write $|X| = |Y|$.

Next, because "finite or same size as $\mathbb{N}$" is so commonly used, I will create a definite for it. I will say that a set X is **countable** if one of the following is true:

1. X is finite, or
2. $|X| = |\mathbb{N}|$.

If you do want to distinguish between the two, the first is said to be **countably finite** and the second is said to be **countably infinite**.

Now for some facts about countability ... be ready for this because when you enter infinity ... things can be pretty weird.

First of all I claim that $\mathbb{N}$ has the same number of things as $\mathbb{N} \cup \{\pi\}$, i.e.,

$$|\mathbb{N}| = |\mathbb{N} \cup \{\pi\}|$$

This seems to be obviously wrong. But remember that counting means finding a 1–1 and onto function. In this case to say that there are as many things in $\mathbb{N}$ as there are in $\mathbb{N} \cup \{\pi\}$, I need to show you a 1–1 and onto function between the two sets. Here you go:

$$\begin{aligned}
\mathbb{N} &\to \mathbb{N} \cup \{\pi\} \\
1 &\mapsto \pi \\
2 &\mapsto 1 \\
3 &\mapsto 2 \\
4 &\mapsto 3 \\
5 &\mapsto 4 \\
\vdots &\quad \vdots
\end{aligned}$$

It should be clear that this function is 1–1 and onto. Therefore

$$|\mathbb{N}| = |\mathbb{N} \cup \{\pi\}|$$

The above is closely related to a very interesting example created by the famous German mathematician David Hilbert called the Grand Hotel ...

You arrive at a hotel, the Grand Hotel (sometimes people call this the Hotel Infinity). The rooms are numbers 1, 2, 3, ... and there are as many rooms as $\mathbb{N}$. Unfortunately the hotel is full. However the hotel owner, because he's really smart (probably because he took automata theory), simply told you to go to room 1 and tell the occupant in room 1 to go to room 2 and tell the occupant in room 2 to move to room 3 and tell the occupant to do likewise, etc. In general, the occupant in room n moves to room $n+1$ and the previous occupant in $n+1$ moves to room $n+2$.

As you can see the Grand Hotel is able to accomodate you even though it's already full Neat right? (Unfortunately, all occupants are annoyed by having to move ...)

Now obviously, this scheme also works if you arrive with a friend, i.e., there are *two* new hotel guests. No problem! The two new guests occupy room 1 and room 2 and ... you know what to do.

But what if the number of new guests is not finite? What if the number of new guests is in fact as numerous as $\mathbb{N}$ itself!!! In terms out that

$$|\mathbb{N} \cup \mathbb{N}| = |\mathbb{N}|$$

Unbelievable!!! Here's how you assign rooms 1, 2, 3, to $\mathbb{N} \cup \mathbb{N}$:

pic

What a happy owner of Grand Hotel!

But what if you have $\mathbb{N} \cup \mathbb{N} \cup \mathbb{N}$ guests? No problemo amigo! (Sorry ...)

pic

Not only that ... in fact $|\mathbb{N} \cup \mathbb{N} \cup \mathbb{N} \cup \cdots| = |\mathbb{N}|$ where the union on the left is a union of countably infinite copies of $\mathbb{N}$. Wow! To be more precise, I would write

$$\left| \bigcup_{n=1}^{\infty} \mathbb{N} \right| = |\mathbb{N}|$$

A union involving countable many sets is called a countable union. The "countable" in "countable union" refers to the number of sets you're trying to "union-up", not the sets themselves.

Theorem 15.2.1. *The countable union of countable sets is countable.*

Instead of cobbling together more and more copies of $\mathbb{N}$, let's take a break and look at other sets. For instance there's $\mathbb{Q}$, the set of fractions. Not surely there are more things in $\mathbb{Q}$ than in $\mathbb{N}$ because $\mathbb{Q}$ contains $\mathbb{N}$ and surely the fraction $1/2$ is not in $\mathbb{N}$. Right?

WRONG!

That's because you can associating values of $\mathbb{N}$ with values in $\mathbb{Q}$ in a wrong way. You can think of $\mathbb{Q}$ as (more or less) $\mathbb{N} \times \mathbb{N}$. For instance $3/7$ corresponds to the point $(3, 7)$ in $\mathbb{N}^2$ But wait a minute ... look at the picture of $\mathbb{N} \times \mathbb{N}$. Doesn't it remind you of $\mathbb{N} \cup \mathbb{N} \cup \mathbb{N} \cup \cdots$? To be more concrete, let me show you how to associate $\mathbb{N}$ with the points in $\mathbb{N} \times \mathbb{N}$:

pic

There are some details you have to be careful about for instance What about negative fractions? The fractions also does not include something like $5/0$ (which is not defined!) However in $\mathbb{N} \times \mathbb{N}$ we do have $(5, 0)$. What about $(3, 6)$ that would correspond to the fraction $6/3$. But this is also the same as $2/1$. All these are not too difficult to overcome.

Note the technique very carefully. In you try to label the points *horizontally* like this:

you will get into trouble because you're ever going to return to start labeling the next row!!!

Your mind hurts, right?

Exercise 15.2.1. Complete the proof of $|\mathbb{Q}| = |\mathbb{N}|$ by explaining what you should do about negative fractions or about points like $(5, 0)$ or about the issue with $(6, 3)$ and $(2, 1)$. □

Now at this point, you might give up and say that $\mathbb{R}$ is probably as big as $\mathbb{N}$. Actually ... $\mathbb{R}$ is strictly bigger than $\mathbb{N}$ in size!!! Phew!!! For a moment ... we thought the whole world is nothing more than $\mathbb{N}$.

The proof that $\mathbb{R}$ is in fact bigger than $\mathbb{N}$ is important because the technique itself will be used later in a totally different context ... in the world of languages and automata.

This is so important that I need a new section!!! ...

15.3 Cantor's Diagonal Trick: Counting $\mathbb{R}$ debug:

cantors-diagonal-trick.tex

Instead of showing that $\mathbb{R}$ is bigger than $\mathbb{N}$, I'm going to show you that in fact the interval of real numbers $[0, 1)$ is already bigger than $\mathbb{N}$. The proof technique was invented by Georg Cantor, the mathematician who defined this whole new way of counting by using 1–1 and onto functions.

First of all let me just declare that I'm going to prove our goal by contradiction. I'm going to assume that $[0, 1)$ is countable. Therefore I can list the values in $[0, 1)$, say

$$x_0, x_1, x_2, x_3, \ldots$$

(The goal is to show you that I can find a value not in the above list.)

Next, I'm going to express the x_i as decimal numbers. For instance if $x_0 = 1/2$, then I'm going to write $x_0 = 0.5$ instead. Furthermore, I'm going to write the numbers with infinitely many decimal places. For instance if $x_1 = 0.5$, I'm going to write

$$x_0 = .5000000000000000\ldots$$

I'm going to give names to the decimal digits of x_0. The i–th digit is written x_{0i} So in this case,

$$\begin{aligned} x_{00} &= 5 \\ x_{01} &= 0 \\ x_{02} &= 0 \\ x_{03} &= 0 \\ x_{04} &= 0 \end{aligned}$$

etc. If x_2 is 0.14159265..., then

$$\begin{aligned} x_{10} &= 1 \\ x_{11} &= 4 \\ x_{12} &= 1 \\ x_{13} &= 5 \\ x_{14} &= 9 \end{aligned}$$

etc. So far so good ... we have a list of numbers $x_0, x_1, x_2, \ldots$ and a bunch of

digits

$$x_{00}, x_{01}, x_{02}, ...$$
$$x_{10}, x_{11}, x_{12}, ...$$
$$x_{20}, x_{21}, x_{22}, ...$$

Let me lay the digits in a grid:

I will fill the cell at row x_i and column j with the x_{ij}. For instance since $x_0 = 0.5000\ldots$ and $x_1 = 0.14159254\ldots$, I have the following:

Now I'm going to construct a value y that is in $[0, 1)$ that is not in the above list. I'm going to build y by specifying the decimals of y. Just like the x_i's above, y is going to look like this:

$$y = 0.y_0y_1y_2\cdots$$

where y_i is the i–digit of y to the right of the decimal point. In order for y to contradict our assumption about the countability of $[0, 1)$, I will need y not to be x_0, and not be x_1, and not be x_2, etc.

To make y not x_0, I look at the first digit of x_0 to the right of the decimal place, i.e., x_{00}. All I need to do is to choose y_0 to be different from x_{00}. For instance if $x_{00} = 5$, I can choose 0 for y_0 (of course I can also choose 1 for y_0 ... any digit that is not 5 works.) In other words I make y *not* x_0 by making them different at the first decimal place.

So at this point y looks like

$$y = 0.0y_2y_3\cdots$$

Now suppose the second digit of x_1 to the right of the decimal point is 1. Using the same trick, I will choose y_1 to be different from $x_{11} = 1$. For instance I can choose $y_1 = 7$. At this point

$$y = 0.07y_3y_4y_5\cdots$$

Etc.

In general, for $i \geq 0$, y_{ii} to be an element in

$$\{0, 1, 2, \ldots, 9\} - \{x_{ii}\}$$

This will ensure that

$$y \neq x_i$$

Note that the form of y tells me that y is indeed in $[0, 1)$. We have found a value, y, in $[0, 1)$ which is not in the list

$$x_1, x_2, x_3, ...$$

which I assumed at the beginning is a complete list of values in $[0, 1)$.

Contradiction!!!

Actually there's a small point: What if I accidentally select $y_0 = 9$, $y_1 = 9$, $y_2 = 9$, ... This means that y looks like $0.9999\cdots$. This value is not in $[0, 1)$. This value is actually 1 and it not in $[0, 1)$. (Right? Check with your math textbooks or with your math instructors.) In the same way $0.499999\cdots$ is actually the same as 0.5. To avoid issues like these, I will just make sure that I will never choose 9 for the y_i's. Problem fixed!!!

This proof technique is called the **Cantor's diagonal trick** (or Cantor's diagonal method if you want to make it sound more respectable.)

Theorem 15.3.1. $\mathbb{R}$ *is not countable.*

Now you might think that $[0, 1)$ is much smaller than $\mathbb{R}$. But in fact ...

Exercise 15.3.1. Show that $|\mathbb{R}| = |[0, 1)|$.

I am now going to show you that there is a language (over a fixed Σ throughout this argument of course, say $\Sigma = \{a, b\}$) which is not accepted by a Turing machine, i.e., the language is not Turing–recognization (or recursively enumerable.)

I'll do this in two different ways: the first way is to show that there are more languages than Turing machines (the method is by counting) and the second way is by defining a language that is not Turing–recognizable.

First of all ... how many Turing machines are there? The number of Turing

machines is actually countable.

Why?

Because each Turing machine can be encoded as a finite binary string. This means that the collection of Turing machines is a subset of

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Note that each of the Σ^n is countable (in fact finite!) Therefore Σ^* is a countable union of countable sets. This implies that Σ^* is countable.

Theorem 15.3.2. *The set of Turing machines (for a fixed Σ) is countable.*

Before we count $\textsc{Lang}_\Sigma$, the collection of languages of Σ, let's try another counting example that involves Cantor's diagonal trick.

Let's count the number of functions from $\mathbb{N}$ to $\{0, 1\}$. An element of this set is of the from

$$f : \mathbb{N} \to \{0, 1\}$$

For convenience, let me call this set X. I claim that X is not countable. By my assumption, I can list the elements in X like this:

$$f_0, f_1, f_2, \ldots$$

f_0 is of course completely determined by

$$f_0(0), f_0(1), f_0(2), f_0(3), \ldots$$

Each $f_i(j)$ is either 0 or 1. Now I'm going to build a function $g : \mathbb{N} \to \{0, 1\}$ which is not in the above list. How?

This is actually very similar to the setup in Cantor's argument above. In this case we have:

where the cell at row f_i and column j is filled with the value of $f_i(j)$. For instance if the table looks like this:

Then $f_0(0) = 1$ and $f_1(2) = 0$, etc.

Well, g is completely determined by

$$g(1), g(2), g(3), ...$$

So I need to specify $g(n)$ for $n \geq 1$. For $g(1)$, I'm going to choose a value for $g(1)$ so that g is not f_1. Well, that's easy ... I follow Cantor's diagonal trick and choose $g(0)$ to be different from $f_0(0)$. If $f_0(0)$ is 1, I will set $g(0) = 0$. If $f_0(1)$ is 0, I will set $g(1) = 1$. Likewise, I choose a value for $g(2)$ so that $g(2) \neq f_2(2)$. Etc.

In general I construct my function g such that for each $n \geq 0$,

$$g(n) = \begin{cases} 0 & \text{if } f_n(n) = 1 \\ 1 & \text{if } f_n(n) = 0 \end{cases}$$

I have constructed a function $g : \mathbb{N} \to \{0, 1\}$ such that $g \neq f_0$, $g \neq f_1$, ... Therefore the list of functions $f_0, f_1, f_2, \ldots$ cannot be complete.

Therefore X is not countable.

Theorem 15.3.3. *The set of functions from* $\mathbb{N}$ *to* $\{0, 1\}$ *is not countable.*

Exercise 15.3.2. Is the collection of subsets of $\mathbb{N}$ countable? □

Exercise 15.3.3. The even integers are having a having a war with the odd integers. (a) Each even integer has 1 cannonball. However many ways can the even integers fire their cannonballs? If it's infinite, then is it countable? (b) What if each even integer has 5 cannonballs? (c) What if each even integer has a finite number of cannonballs? (The the number can be diffrent? (d) Countable number?

Exercise 15.3.4. Assume that all (undirected) graphs with finitely many nodes have their nodes labeled using $\mathbb{N}$. Is the collection of such graphs countable? □

15.4 Turing Machines debug: informal.tex

Informally, a Turing machine (TM) is like a DFA or a PDA except for the following:

- For the DFA (and PDA), the machines read the input string one character at a time from the leftmost character to the rightmost. A TM can also read the input one character at a time. However the input string should be thought of as being written on an infinitely long tape; you should think of the tape as being padded infinitely to the right with blanks. The TM can actually write to it as well. Therefore you should think of the TM as having a read/write head that moves along an infinitely long string. Furthermore, the read/write head can move either left or right or it can also stay. The only restriction is that it must stay on the infinite tape.
- For the DFA, acceptance is determined by what the state the machine is in once the string is completely read. For the TM, since the input string is written on an infinite tape, there is really no end-of-string. Therefore for a TM, a string is accepted if the machine enters an accepting state, regardless of the position of the read/write head.

Even without a formal definition of a TM, you probably know by now that the most important thing about running the TM is that you need to know what happens when the machine is in a particular state and it is about to read a particular character. What is the machine capable of doing? Well, obviously it has to go to another (possibly the same) state. It can overwrite the character it is reading; this is sort of like the PDA. Furthermore it can move it's read/write head either to the left or right or it stays. So the transition function should look like:

$$\delta(q, a) = (p, b, D)$$

where q and p are states, a and b are the allowable characters on the infinite tape and D refers to the direction moved by the read/write head. D can take values L (for left), S (for stay) and R (for right).

15.5 Formal Definition of TM debug: formal.tex

Here's the formal definition. Most of this won't be a surprise to you by now. There's one thing I should mention. We will distinguish between the alphabet of the string used to generate a language (the Σ) and the alphabet that the TM can write on the tape.

Definition 15.5.1. A (deterministic) **Turing machine** TM M is made up of

- Q: a finite set of states.
- Σ: a finite set of input alphabet.
- Γ: a finite set of alphabet the TM using for read and write. Σ is a subset of Γ. Furthermore the blank (or space) character $\sqcup$ is also in Γ.
- q_0: the initial state. This is in Q.
- F: the set of accepting states. F is a subset of Q.
- R: the set of reject states. R is a subset of Q.
- B: the blank symbol; this is in Γ but not in Σ. In most TMs (and textbooks), this character is standardized and the same, so it's usually not specified when describing a TM.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$ is the transition function.

Here's how you draw a TM given the above specification: Everything is the same as a DFA. For the transitions, suppose you have

$$\delta(q, a) = (p, b, R)$$

Then you draw a directed edge or transition from state q to state p and label it $a/b, R$ or $a \to b, R$

To be formal about computation we will introduce the following notation:

Definition 15.5.2. Let

$$x_1 \dots x_m q x_{m+1} \dots x_n$$

to denote that fact that the TM current has it's read/write head pointing to the $(m+1)$-st character on the tape x_{m+1}; this will be the *next* character the TM will read. This expression

$$x_1 \dots x_m q x_{m+1} \dots x_n$$

is called an **instantaneous description** (ID). In some books this is also called a **configuration**. Of course if you're running the TM with string x, the initial ID is

$$q_0 x$$

Of course these are all just notation for description a computation of the TM. In particular if $\delta(q, a) = (b, p, R)$, then

$$xqay \vdash xbpy$$

The read/write head over-writes a with b and moves right. Make sure you understand that! If instead of that you have $\delta(q, a) = (b, p, L)$, then

$$xcqay \vdash xpcby$$

so that the character being read, a, is over-written by b and the read/write head moves left. And if the transition if $\delta(q, a) = (b, p, S)$, then the read/write head stays:

$$xqay \vdash xpby$$

As always, if we will use $\vdash^*$ to note "the same or at least one derivation".

Exercise 15.5.1. Prove that

1. $\vdash^*$ is reflexive
2. $\vdash^*$ is transitive

□

Exercise 15.5.2. Implementation note: Of course instead of using

$$wqw'$$

to describe an instantaneous description (where w, w' are words and q is a state), one can also use

$$(w, q, w')$$

In terms of implementation, this is clearly better. Otherwise in your program, you would have to continually search for the state within wqw' ... which is annoying.

Definition 15.5.3. $x \in \Sigma^*$ is **accepted** by M is there is some $p \in F$ such that

$$q_0 x \vdash^* ypz$$

for some strings y, z in Γ^*. And, surprise-surprise, $L(M)$ is the set of strings accepted by M.

There are many minor variations of the TM definition and they are all essentially the same. For instance:

1. Instead of having a set of accept states F, it's possible to redefine the TM to have one single accept state without changing what the TM does. In this case, the single accept state is usually written q_{accept}.
2. Likewise it's possible to redefine a TM to have only one reject state. In this case the special reject state is written q_{reject}.
3. It's possible to rewrite the TM so that the read/write head does not stay but always either move left or right. In this case, the transition function is of the form:
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

Definition 15.5.4.

- It's easy to show that in fact you only need one accepting state. (Why?) Therefore some books will just have a special state called q_{accept} for the only accepting state of the TM.
- Note that if the TM tries to move to the left while it's already at the leftmost position on the tape, then the machine crashes and stops running. The string is not accepted. Instead of allowing this to happen, some books will define a special state q_{reject}. Whenever the TM enters the q_{reject} state it stops (or halt) and the string is not accepted.
- From the above, the TM can reject by either entering the state q_{reject} or by moving left on the leftmost position on the tape. It's easy to see that you really do not need both cases. You can rewrite the TM so that if it tries to fall off the left edge, you make the TM go into q_{reject} instead. Here's how you do it. Before running the TM, shift all the checks of the input string to the right by one, and insert a "beginning of tape marker"; you can use any symbol not used for instance a common symbol in some books is '$'. Then include a transition so that is the TM is reading the "beginning of tape marker" and it attempts to move the left, replace it with a transition that enters q_{reject} instead. This will stop the machine before it falls off the left edge.
- Although according to the above definition of a TM there is a transition for every state q and every symbol a in Γ, it is customary not to include transitions that enters the state q_{reject}. Therefore in some books some transitions are left out and the authors say that "if the TM is in a state q and is reading a character a but there is no applicable transition, the machine crashes and the string is rejected." This is just the same as

including a transition from such a state and symbol to the rejecting state and of course you cannot exit the rejecting state.

Definition 15.5.5. A language is said to be **recognizable** or **Turing recognizable** or **recursively enumerable** (r.e.) if it is accepted by a TM.

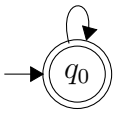
(Note: Computer Science is so new that many concepts still have multiple names. Tune in after 100 hundred years to find out which name finally gets picked.)

Note one curious feature of the TM. It is possible for it to run forever. That's not the case for either the DFA, NFA, or PDA. In particular it's possible for the string not to accepted by the TM entering the q_{reject} state, the TM crashes by moving to the left while at the leftmost position on the tape, or it does not enter an accepting state.

Definition 15.5.6. A language is said to be **decidable** or **Turing decidable** or **recursive** (rec) if it is accepted by a TM that always halts. A TM **always halts** if giving any input string, the TM will either reach an accepting state or the rejecting state. The accepting state and rejecting state are called **halting states**.

Example 15.5.1. It' easy to design a TM that accepts all strings with characters from $\{a, b\}$:

$$(a \to a, S), (b \to a, S), (\sqcup \to \sqcup, S),$$



Example 15.5.2. Let $L = \{a^n b^n c^n \mid n \geq 0\}$. We know that L is not a CFL. We will now prove that it is recursively enumerable, i.e., accepted by a TM.

TMs tend to large. So to simplify the specification of this TM, transitions entering the q_{reject} state is not listed.

The idea is simple: Scan left and right, marking a single a and a single b and a single c in each right scan. We will need special characters to denote that a,b and c are marked. We will use A, B and C respectively. Of course if we cannot find a b or c to match this a, the string is rejected. Now once c is marked, we move all the way to the left to look for the first marked a (i.e., A). Once that's found, we move right. This would either be another a or B. If it's a, we repeat the process of marking a,b,c.

On the other hand if it's B, then there are no more a's. Then we just move right over all the marked characters (i.e., B and C) until we see a blank and accept the string. Of course if b or c is found, then the string is rejected; there shouldn't be anymore b's or c's for the string to be accepted.

That's the main idea. Here's the TM. We will use $\sqcup$ to denote the blank character.

(q, x)	$\delta(q, x)$	
(q_0, a)	(q_1, A, R)	Start marking phase: Mark a and move right
(q_1, a)	(q_1, a, R)	Fast forward over a
(q_1, B)	(q_1, B, R)	Fast forward over B
(q_1, b)	(q_2, B, R)	Mark b and move right
(q_2, b)	(q_2, b, R)	Fast forward over b
(q_2, C)	(q_2, C, R)	Fast forward over C
(q_2, c)	(q_3, C, L)	Mark c and move left
(q_3, C)	(q_3, C, L)	Rewind over C
(q_3, B)	(q_3, B, L)	Rewind over B
(q_3, b)	(q_3, b, L)	Rewind over b
(q_3, a)	(q_3, a, L)	Rewind over a
(q_3, A)	(q_0, A, R)	On seeing A, go to state q_0
(q_0, B)	(q_4, B, R)	Start scanning phase: read first B and move right
(q_4, B)	(q_4, B, R)	Fast forward over B
(q_4, C)	(q_5, C, R)	Read first C and move right
(q_5, C)	(q_5, C, R)	Fast forward over C
$(q_5, \sqcup)$	$(q_{\text{accept}}, B, S)$	Accept string when a $\sqcup$ is found

Make sure you draw the state (or transition) diagram. Again transitions which are not labeled go to the q_{reject} state. Instead of writing infinitely many blanks, we will write one blank just beyond *abc*. If we need to we can add blanks when we need to (actually, for this TM only one blank is needed.)

Now let's trace the execution of M for the string *abc*.

$$\begin{aligned} q_0abc\sqcup &\vdash Aq_1bc\sqcup \\ &\vdash ABq_2c\sqcup \\ &\vdash Aq_3BC\sqcup \\ &\vdash q_3ABC\sqcup \\ &\vdash Aq_0BC\sqcup \\ &\vdash ABq_4C\sqcup \\ &\vdash ABCq_5\sqcup \\ &\vdash ABCq_{\text{accept}}\sqcup \end{aligned}$$

Now try to trace the execution of the TM for *aabbcc* and *aabbccc*.

[TODO: Check if \$ is better for begin-of-tape marker. Might be easier for students to remember if \$ is used for both begin-of-tape for TM and bottom of stakc for PDA. Check if ! can be used for end of tape.

15.5.1 Single accept state

In the formal definition of a TM, one can have any number of accept states. In fact, you don't change the power of the machine if you assume that there's exactly one accept state. When there's one accept state, it's conventional to denote this state as q_{accept}.

Exercise 15.5.3. Let M be a TM with a set of accept states F. Design another TM M' with exactly one accept state such that $L(M) = L(M')$. □

15.5.2 Single reject state

In the formal definition of a TM, one can have any number of reject states. In fact, you don't change the power of the machine if you assume that there's exactly one reject state. When there's one accept state, it's conventional to denote this state as q_{accept}.

Exercise 15.5.4. Let M be a TM with a set of accept states F. Design another TM M' with exactly one reject state such that $L(M) = L(M')$. □

Exercise 15.5.5. Let M be a TM with a set of accept states and a set of reject states. Design another TM M' with exactly one accept state and exactly reject such that $L(M) = L(M')$. □

15.5.3 Read/write head that does not stay put

In our definition of a TM, we allow a read/write head to move right, left, or stay (put). In Sipser's book, his definition does not allow the read/write head to stay (put). It seems that our TM has more power. But in fact:

Exercise 15.5.6. Let M be a TM whose read/write head can move left, right, or stay. Design another TM M' such that that read/write head only move left or right and furthermore $L(M') = L(M)$. □

15.5.4 Moving off the left-end of the tape: begin-of-tape marker

Note that our definition of a TM stipulates that when the read/write head moves to the left on the first cell of the tape, i.e., it attempts beyond to move outside the tape (on the left), the TM "crashes" meaning to say that the TM halts and does not accept the input that it's running on.

In Sipser's book, he defines his TM so that when a TM attempts to move left at the first cell, the read/write head bounces back to the first cell and continues execution.

Let TM be our class of all TMs and let TM$'$ be the class of TMs from Sipser's book in the sense that the read/write head will bounce back when it attempts move to the left from the first cell.

Exercise 15.5.7.

1. Let M in TM. Design a M' in TM$'$ such that $L(M') = L(M)$.
2. Let M' be in TM$'$. Design a M in TM such that $L(M) = L(M')$.

□

Example 15.5.3. Here's a trick. To prevent the case where the TM moves left while it's already at the leftmost position, you need to shift all the character one position to the right. Suppose the symbol `$` is not used. Suppose the TM is M. How would you write a new TM that will shift the input by one space to the right, put `$` as the first character, and move the read/write head so that it's about to read the first character of the input string (i.e., first character to the right of `$`. Once this pre-processing step is done, the new TM starts simulating the old one. In other words in terms of ID, you want a TM M' that will do the following: Suppose the new start state is q_0'.

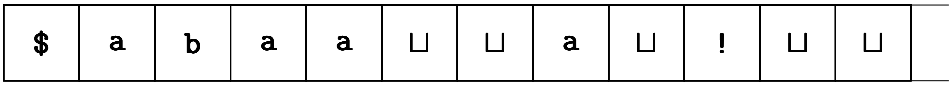
$$q_0 x \vdash \$q_0 x$$

Since this modification can always be carried out on any TM, in many books, it's assumed that the TM will never fall off the left edge. In other words, these authors assume that it the TM tries to move left on the leftmost spot, it will simply stay and not move.

15.5.5 Output; transducer

So far, we use a TM, M, to describe a language, i.e., given a string x, on running M on x, we are interested if M accepts (i.e., M enters an accept state) or M rejects (i.e., M enters a reject state or runs without stopping).

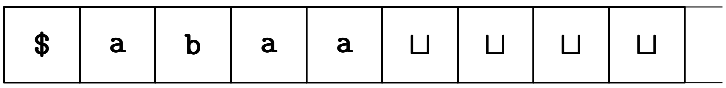
We can also use a TM to produce an output string. This is the string on the input tape when the TM halts. (This means that an output makes sense only when the TM enters an accept or a reject state, not when it runs on and on without stopping.) In the case when the TM uses a begin-of-tape character, `$`, then this `$` is not part of the output. If you use the end-of-tape character `!` to mark the end of output, `!` is not considered part of the output. For instance if the TM halts with this on the input tape:



then the ouptut is

abaa␣␣a␣

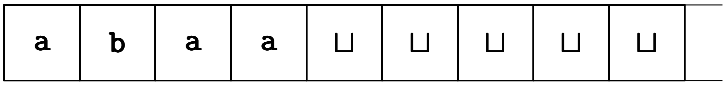
Depending on the book you read, sometimes the transducer will have a set of output characters which does not include the space ␣. In this case, the output will be all the characters up to but not including the first space. For instance



then the ouptut is

abaa

In fact, it's easy to modify the transducer so that once the output is written, it shifts the non-space characters to the left by one, removing the begin-of-tape character `$` and it's also easy to replace the end-of-tape marker `!` by a space. For instance in the input tape above, after shifting left by one and replacing the `!` by a space, we get



and then the transducer halts. The output is then, again, the following:

abaa

When a TM is used for input/output, the TM is also called a **transducer**. In the situation when a transducer does not halt, the output is undefined.

It's common to have a symbol to denote the output function. For instance, say M is the transducer, you can say "Let f_M be the output function of M." Suppose in the above, the input for M was `ab`. Then we write

$$f_M(\texttt{ab}) = \texttt{abaa}$$

as you would expect. In terms of derivations of IDs, we have

$$q_0\texttt{ab} \vdash^* f_M(\texttt{ab})q_{\text{accept}}$$

assuming that when M halts, the read/write head is about to read the space after `abaa`.

By the way, for the case when you're using a TM as a transducer, the important thing is the output and not whether the input was accepted or not. In other words, once the output is written on the input tape, you can enter the accept state or the reject state – it doesn't matter which one. So for the case of a Turing transducer, some books will give such a Turing machine a single halting state q_{halt} that plays the role of q_{accept} and q_{reject}.

If a transducer M always halts, then the output function is said to be a **total** function. Otherwise (i.e., if M can run forever so that the output is undefined), we say that output function is a **partial** function.

total

15.5.6 Computation of $\mathbb{N}$

Example 15.5.4. You can use TM to compute numbers. For instance you can use the string 111 to represent 3. 0 is represented by ϵ. Given this data format, you can define the output of a TM M by what's on the tape when then machine halts. Can you define a TM that computes the twice function. In other words $M(111) = 111111$, i.e., if you put 111 on the tape and run the machine, it should halt with 111111 on the tape.

The translation of a mathematical notation in our world into one suitable as input to a TM is called an encoding. So I would say that 11111 is an encoding of 5. It's common to use $\langle x \rangle$ to denote an encoding of x. So I would write for instance $\langle 5 \rangle = 11111$.

If the TM does not halt on input x, then we say $M(x)$ **diverges**. [PUT THIS SOMEWHERE ELSE.]

- You can design a TM to add numbers. For instance to add two 3 and 5, the input to the TM is 111011111, i.e., $1^3 01^5$. The expected output is 11111111, i.e., 1^8. Can you design such a TM? In general, can you design a Turing machine (a transducer) M such that if f_M is the output function of M, then
$$f_M(1^m 01^n) = 1^{m+n}$$
- Can you design one to perform subtract of the form $m-n$ where $m \geq n$? In other words you want to design a TM M such that if g_M is the output function, then $g_M(1^m 01^n) = 1^{m-n}$.
- Can you design a TM to do multiplication? For instance to compute the product of 3 and 4, you run the machine with 11101111 and get 111111111111.
- Can you design a TM M that can computer powers of 2? For instance $M(1) = 11$, $M(111) = 11111111$.

You can of course combine all the above features into a single TM. For instance you can use 1 to denote $+$, 11 to denote $-$, and 111 to denote $*$. In that case to compute $10 + 15$, you use the input $1^{10}0101^{15}$; to compute $10 - 5$, you use $1^{10}01101^5$; to compute $10 * 15$, you use $1^{10}011101^{15}$.

Exercise 15.5.8. Now use 1111 to denote the integer quotient and 11111 to denote remainder. Add these features to your TM for operations on $\mathbb{N}$. □

Exercise 15.5.9. Add exponentiation to your TM. □

Exercise 15.5.10. The natural thing to do next is $\mathbb{Z}$. We can encode value of $\mathbb{Z}$ if we have a "sign". For instance for 5, we can use the encoding 11^5. For -5, we can use encoding 01^5. In other words we use 1 for "positive" and 0 for "negative". Build an TM that works like an integer calculator that supports the obvious operators. □

Exercise 15.5.11. (Binary representation) If we allow our Σ to have more characters, we can use a different encoding. For instance, say $\Sigma = \{0, 1, \#\}$. Then we can use # as a "separator". For instance to encode $5 + 7$, I can use

$$1^5\#1\#1^7$$

This frees up 0. Therefore I can use binary number representation. For instance 5 now can be encoded as 101 (i.e., 101_2). In that case $5 + 7$ becomes

$$\langle 5\rangle\#\langle +\rangle\#\langle 7\rangle = 101\#1\#111$$

Using this encoding to design a TM that computes $\mathbb{Z}$. □

Exercise 15.5.12. Of course you can also compute rationals. For instance you can encode the fraction $-3/4$ by

$$\langle -3\rangle 0\langle +4\rangle = \langle -\rangle\langle 3\rangle 0\langle +\rangle\langle 4\rangle = 01^3011^4$$

Can you design a TM to add rationals? □

Example 15.5.5. Design a TM that accepts $\{a^nb^nc^n \mid n \equiv 1 \pmod 4\}$. The solution is easy. First the TM checks the input is of the form $a^nb^nc^n$. (We've already done that). After this point if the a's might be marked with another character. Let's say it's been replaced by A's. First the TM go to the leftmost A. It moves right so that the read/write head is reading the next A (or possibly B). It will then continually read 4 A's until a B is reached. If that can be done, then TM enters the q_{accept} state. Otherwise it enters the q_{reject} state.

There are many other computational models (λ–calculus random access machines, general recursive functions, etc.) But in the end they are all proven to be only as powerful as the original Turing machines. Therefore solving a problem algorithmically is believed to be equivalent to solving a problem using a Turing machine. This is known as the **Church-Turing thesis**.

15.5.7 Beginning-of-tape

It's convenient to have a special character to mark the beginning of the input tape. This is useful for instance if you want to "rewind" the read-write head to the beginning before processing the input tape from left-to-right. I will use $ to denote the beginning-of-tape character. This is a character in Γ but not in Σ.

In the Sipser book, he has the assumption that "when the read-write head attempts to move left at the first square of the input tape, then it does not move."

With the beginning-of-tape character $ at the beginning of the input tape just before the input, we can simulate the behavior of Sipser's TM by adding this transition:

$$\delta(q, \$) = (q, \$, R)$$

for all $q \in Q$.

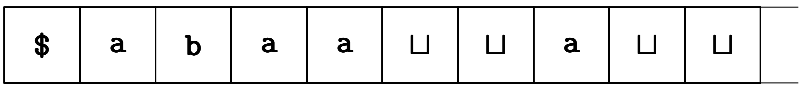
Exercise 15.5.13. Let M be a DTM with no $ provided automatically as beginning-of-tape marker. Construct a new DTM from M, say M', so that at the beginning (that means at the start state), the M' "shifts" the input to the right by one square, inserts $ as the first character, this new M' has the transition

$$\delta(q, \$) = (q, \$, R)$$

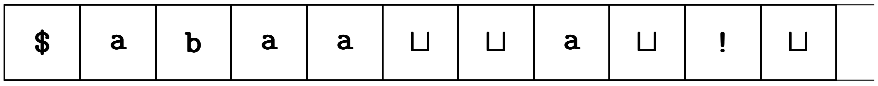
for all $q \in Q$ and otherwise M' "work the same as" M. □

15.5.8 End-of-tape

Besides having the beginning-of-tape marker `$`, we can also have an end-of-tape marker `!`. (This is my name.) The end-of-tape marker is meant to mark where the read-write head has reached. It does not mean that the tape is finite. For instance, say you have the following input tape (with beginning-of tape marker):



and the read-write head actually reached the space after the `a` furthest from the `$`, then this is what the input tape looks like when it has an end-of-tape marker:



The end-of-tape marker is useful if I want to move the chunk of characters that is processed so far to the right by say one square.

Exercise 15.5.14. Check that if M is a DTM with beginning-of-tape marker, then it can be converted to a DTM with beginning–and–end–tape marker. □

Exercise 15.5.15. Given a deciding TM M, construct a DTM M' such that $L(M) = L(M')$ and for each $w \notin L(M')$, M' does not halt. □

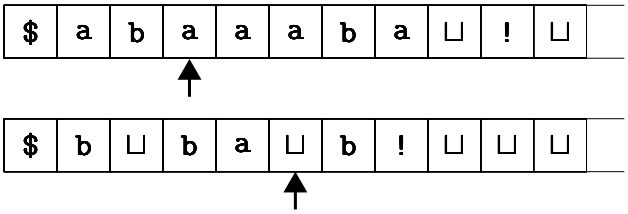
15.6 Multi-tape TM debug: multitape.tex

Consider a version of the TM where there are multiple tapes. The input is placed on the first. The TM has one read/write head for each tape. (The read/write head for each tape can even have the ability to read/write a different set of characters.) Say this TM has k read/write heads (and k tapes). Then the transition function is of the form

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

The input for such a k–tape machine is placed on the first tape.

For instance suppose there's a 2-tape TM that is currently at state q and the two input tapes look like this:

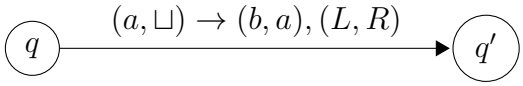


(For readability, I've assumed the TM mark the end-of-tape for each tape.)

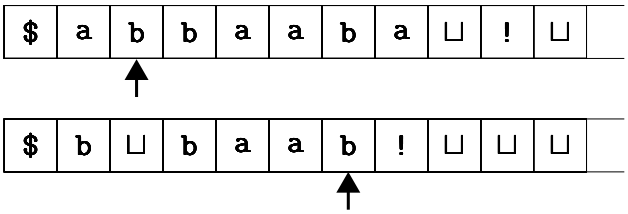
Note that there are two read/write heads (look at the arrows above), the first read/write head is about to read the fourth character on the first tape and the second is about to read the sixth character on the second tape. Suppose the relevant transition is

$$\delta(q, (a, \sqcup)) = (q', (b, a), (L, R))$$

i.e., this is part of the TM diagram:

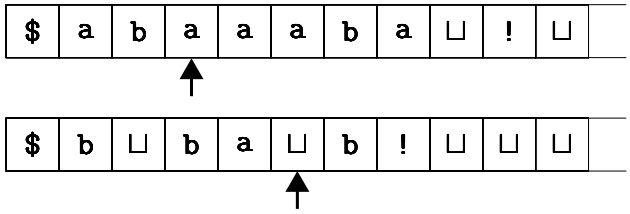


On running the transition the TM is at state q' and the tapes look like:

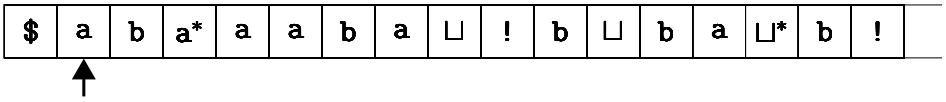


Is this machine more powerful? No. You can prove that it's as powerful as the "regular" TM. In other words given a k–tape TM M, there is a 1–tape TM M' such that $L(M) = L(M')$.

As an example, if a 2-tape TM has this tape configuration:

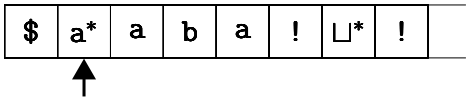


the 1-tape regular TM can simulate the above with this tape:



Note that the asterisk in $\mathtt{a}^*$ in the regular TM records the fact the for the corresponding 2-tape TM, the first read/write head is about to read this a. Nice idea right? In general for each character in the Γ of the 2-tape machine, there are two characters in the Γ of the 1-tape machine. So for the above example the 2-tape TM has $\Gamma = \{\mathtt{a}, \mathtt{b}, \sqcup\}$ and the Γ of the corresponding 1-tape TM is $\{\mathtt{a}, \mathtt{a}^*, \mathtt{b}, \mathtt{b}^*, \sqcup, \sqcup^*, \$\}$

Here's how this TM M' works. If initially the input is say *aaba* for the 2-tape M (on the first tape), M' will initialize the tape this way:



Suppose M is at state q. M' will then scan left to locate the marked characters Once it reaches the second !, it knows that the two characters that the corresponding 2-tape machine is looking at. Say it's (c, c'). Then M' "remember" that it should be at state q and should be reading character (c, c'). In other words M' enters state (q, c, c') – the new TM M' that simulates M must have such states. The TM M' now "knows" the current state of computation of M.

Based on how the original M works, i.e., based on the transition of M at state q and reading (c, c'), the corresponding machine M' now scans left and simulate what M should do, modify the input tape as it scans.

I'll leave the rest to you – there are lots of details to work out, but it's not

difficult once you get the idea of merging multiple tape segments onto one long tape.

15.7 Bidirectional Tape TM debug: bidirectional.tex

Instead of a TM with the infinite tape going one way, what if we consider a TM with a bidirectional tape? In other words the tape goes infinitely to the left *and infinitely to the right* and before you run the machine, the input is placed in the "middle" and the read/write head starts at the first character of the string. Is this TM more powerful? No! Because it can be simulated with a 2-tape TM which you know can be simulated with the usual TM.

15.8 RAM - random access machine debug: ram.tex

Our TM moves on its input tape relatively: it moves to the left or right or stay with respect to its current position. What if we allow the TM to go to the memory cell at any index position? (This is like a modern computer accessing a byte in its RAM at any address location.) Specifically, this TM has another tape – I'll call it the register tape. It contains the index that the read-write head should go to. This means the following: Suppose the random access machine has its read-write head at index position n and it's at state q. Furthermore the character at index n of the input tape is c. If the transition is

$$\delta(q, c) = (q', c')$$

this means that the character c at index n is ovewritten by character c', the machine enters state q' and the read-write goes to cell at index position n. Initially, the read-write head is at index 0.

Let $\mathrm{TM}_{\mathrm{RAM}}$ denote the class of such machines.

Exercise 15.8.1.

1. Let $M \in \mathrm{TM}$. Construct $M' \in \mathrm{TM}_{\mathrm{RAM}}$ such that $L(M') = L(M)$.
2. Let $M' \in \mathrm{TM}_{\mathrm{RAM}}$. Construct $M \in \mathrm{TM}$ such that $L(M) = L(M')$.

□

More generally, a random access machine can have more than one, but a finite number of, register tapes. In that case the transition would be $delta(q, c) = (q', c', i)$ where the position the read-write head should move to is on register tape number i.

The above is my simplification of a Turing random access machine. The full blown random access machine is more like a modern computer.

15.9 Nondeterministic Turing Machines debug: ntm.tex

From our definition of TM and also from your experience from DFA, you would expect a nondeterministic version of a TM. In our definition of a (deterministic) TM, for each state q and $a \in \Gamma$, there is a corresponding new state q' and a symbol a' you overwrite a with and also there is a direction D taken by the read/write head (D is either L,R or S):

$$\delta(q, a) = (q', a', D)$$

Recall as stated above sometime we do allow $\delta(q, a)$ to be undefined. In this case, the machine crashes. This is equivalent to saying that $\delta(q, a) = q_{\text{reject}}$. As mentioned earlier TM are so huge that usually authors omit such description of δ with the understanding that the transition leads to a rejecting state. Therefore with this convention, $\delta(q, a)$ is either defined or not.

The definition of a nondeterministic TM (NTM) is easy. You allow multiple options for $\delta(q, a)$. In other words $\delta(q, a)$ is a subset of $Q \times \Gamma \times \{L, R, S\}$, i.e.,

$$\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L, R, S\})$$

Of course if an NTM has a δ satisfying $|\delta(q, a)| = 1$, then it is really just a deterministic TM.

Given a NTM N and a string x, we say that N accepts x is there is a sequence of IDs that leads to an accepting ID. Note that just like the case of the NFA, there might be several choices at some time. To accept a string, you only need to have *one* accepting computation; the others can fail.

And ... (surprise, surprise) ...:

Theorem 15.9.1. *Deterministic TM is just as powerful as nondeterministic TM. In other words:*

(a) If M is a TM, then there is an NTM N such that $L(M) = L(N)$.
(b) If N is an NTM, then there is a TM M such that $L(N) = L(M)$.

Here's a sketch of the proof. (a) is easy and is already hinted upon above. What about (b)? Suppose you're given N. You need to design a TM M that can try out all possible computational sequence of N. How would you do that? Suppose

$$r = \max_{q,a} |\delta(q, a)|$$

where q, a runs over all the states in Q and all symbols in Γ of N. Suppose you want to run N on string x. The initialize configuration is q_0x. There can be at most r possibilities. Basically the new M runs track of all possible computations of the "clones" on a single tape with the clone separated by an appropriate separator character. The computations are of course recorded using IDs. The TM will need to divide time equally among all the clones! In other words, if there are 10 ongoing computations, the TM must keep track of which computation is currently active, and execute one step for each clone and move on to the next. The reason why this is necessary is because one particular clone might run on forever. So you can't just put all your time into one clone! This is important.

TODO:

- Mention given a TM M, it's possible to design another TM M' that simulates M executing an ID for one step. The input is the $\langle ID \rangle$ and after executing this in one step, M' halts.
- It's possible to design U such that given $\langle M, ID \rangle$, it runs M on the ID by one step so that the input tapes becomes $\langle M, ID' \rangle$

15.10 Generators debug: generators.tex

A TM M is a generator if it has several "work" tapes and a special "output" tape. This is how the TM is used. All tapes are initially blank. You run the TM without any input. While the machine is running, you observe the output tape. At any point in time, if you see the string $\#w\#$ where $\#$ is a special delimiter and w is a string (the second $\#$ basically is an end–of–string marker to tell us the output for the current string is done), then you say that the M has just generated w. We will write $G(M)$ for the set of strings generated by M. Note there is no particular order in the strings being generated. Also, some strings can be generated more than once.

Note that this is not more powerful than the regular TM.

Theorem 15.10.1.

1. *If M is a TM, then there is some generators M' such that $G(M') = L(M)$.*
2. *If M' is a generator, then there is some TM M such that $L(M) = G(M')$.*

15.11 Languages which are not Turing Recognizable

debug: not-turing-recognizable.tex

After so much work to build more and more powerful machines from DFA to PDA and then to Turing machines, it's a let down that there are still languages that we cannot compute. For a fixed Σ say $\Sigma = \{0, 1\}$, there are still languages not accepted by a Turing machine.

I'll show you why.

The first proof does not really construct a language which is not Turing recognizable. We're also not using any kind of "pumping lemma for Turing machines". Instead I am going to "count" the number of Turing machines and the number of languages and then show you that there are more languages than Turing machines.

Before we do that, I will need to expand your mind a little ... you see there are infinitely many Turing machines and infinitely many languages! We have to "count" in the world of infinities and I have to say (in a sense) that the infinity of the Turing machines is smaller than the infinity of languages.

So we have to go back to the days when you first were born ... and to count all over again ...

15.12 Back to Basics ... Counting (a la Cantor)

debug: cantor.tex

Let's begin by reviewing what we mean by counting. When I give you the following:

A, B, C, D, E

you should tell me that you are seeing 5 things. Why is that? Because since you started to learn, you were told to associate 1, 2, 3, ... to the things you see. So for instance in the above, you would do put your finger on A and say "one", move your finger to B and say "two", etc. That's the same as doing this:

A	B	C	D	E
↑	↑	↑	↑	↑
1	2	3	4	5

i.e., associating the number 1 to A (instead of associating your verbal "one" to A), etc. This association (think "function") is 1–1 and onto (if you were taught to count correctly!) If it's not 1–1, it might be something like this: you might have for instance 1 and 2 pointing to A, you would be counting A twice. If the association is not onto, you would be missing some letter.

Another way to think of counting the above is there you're finding a 1–1 and onto function from the set

$$\{1, 2, 3, 4, 5\}$$

to the set

{A, B, C, D, E}

Therefore counting

A, B, C, D, E

is the same as trying to a 1–1 and onto function from

$$\{1\}$$

to {A, B, C, D, E}, or from

$$\{1, 2\}$$

to $\{A, B, C, D, E\}$, or from

$$\{1, 2, 3\}$$

etc. until you succeed. In our case we succeed when we use

$$\{1, 2, 3, 4, 5\}$$

You can think of

$$\{1\},\ \{1, 2\},\ \{1, 2, 3\},\ \{1, 2, 3, 4\},\ \{1, 2, 3, 4, 5\},\ \{1, 2, 3, 4, 5, 6\},\ \{1, 2, 3, 4, 5, 6, 7\}, ...$$

as our standard measurements for counting sort of like standard weights for weighing things. But why should we use 1–1 and onto functions and these sets as a way to define counting?!? Counting is so basic and such a primitive concept, why bother trying to view counting as something involving sets, functions, and 1–1 and onto functions?!? Shouldn't "counting" be simpler than "1–1, onto functions"???

Well, the reason is because this definition of counting uses sets ... and *sets can be infinite*. For instance I can use the whole set of natural numbers to count the number of things in the set X:

$$\{0, 1, 2, 3, ...\} \to X$$

and the standard measuring set $\{0, 1, 2, 3, ...\}$ is not finite!!! (Remember: depending on who you talk to the set $\mathbb{N}$ might or might not contain 0.)

Therefore if you want to say that two sets X and Y are different, one way would be to say that X has as many things as $\mathbb{N} = \{0, 1, 2, 3, ...\}$ while Y has as many things as $\mathbb{R}$ ... that is if you know that $\mathbb{R}$ is "bigger" than $\mathbb{N}$ which I will show you in a minute.

First of all let me say that two sets X and Y are equinumerous (or informally, they have the same size) if there is a 1–1 and onto function from X to Y. In that case I will write $|X| = |Y|$.

Next, because "finite or same size as $\mathbb{N}$" is so commonly used, I will create a definition for it. I will say that a set X is **countable** if one of the following is true:

1. X is finite, or
2. $|X| = |\mathbb{N}|$.

If you do want to distinguish between the two, the first is said to be **countably finite** and the second is said to be **countably infinite**.

Now for some facts about countability ... be ready for this because when you enter infinity ... things can be pretty weird.

First of all I claim that $\mathbb{N}$ has the same number of things as $\mathbb{N} \cup \{\pi\}$, i.e.,

$$|\mathbb{N}| = |\mathbb{N} \cup \{\pi\}|$$

This seems to be obviously wrong. But remember that counting means finding a 1–1 and onto function. In this case to say that there are as many things in $\mathbb{N}$ as there are in $\mathbb{N} \cup \{\pi\}$, I need to show you a 1–1 and onto function between the two sets. Here you go:

$$\begin{aligned}
\mathbb{N} &\to \mathbb{N} \cup \{\pi\} \\
1 &\mapsto \pi \\
2 &\mapsto 1 \\
3 &\mapsto 2 \\
4 &\mapsto 3 \\
5 &\mapsto 4 \\
\vdots &\quad \vdots
\end{aligned}$$

It should be clear that this function is 1–1 and onto. Therefore

$$|\mathbb{N}| = |\mathbb{N} \cup \{\pi\}|$$

The above is closely related to a very interesting example created by the famous German mathematician David Hilbert called the Grand Hotel ...

You arrive at a hotel, the Grand Hotel (sometimes people call this the Hotel Infinity). The rooms are numbers 1, 2, 3, ... and there are as many rooms as $\mathbb{N}$. Unfortunately the hotel is full. However the hotel owner, because he's really smart (probably because he took automata theory), simply told you to go to room 1 and tell the occupant in room 1 to go to room 2 and tell the occupant in room 2 to move to room 3 and tell the occupant to do likewise, etc. In general, the occupant in room n moves to room $n+1$ and the previous occupant in $n+1$ moves to room $n+2$.

As you can see the Grand Hotel is able to accomodate you even though it's already full Neat right? (Unfortunately, all occupants are annoyed by having to move ...)

Now obviously, this scheme also works if you arrive with a friend, i.e., there are *two* new hotel guests. No problem! The two new guests occupy room 1 and room 2 and ... you know what to do.

But what if the number of new guests is not finite? What if the number of new guests is in fact as numerous as $\mathbb{N}$ itself!!! In terms out that

$$|\mathbb{N} \cup \mathbb{N}| = |\mathbb{N}|$$

Unbelievable!!! Here's how you assign rooms 1, 2, 3, to $\mathbb{N} \cup \mathbb{N}$:

pic

What a happy owner of Grand Hotel!

But what if you have $\mathbb{N} \cup \mathbb{N} \cup \mathbb{N}$ guests? No problemo amigo! (Sorry ...)

pic

Not only that ... in fact $|\mathbb{N} \cup \mathbb{N} \cup \mathbb{N} \cup \cdots| = |\mathbb{N}|$ where the union on the left is a union of countably infinite copies of $\mathbb{N}$. Wow! To be more precise, I would write

$$\left| \bigcup_{n=1}^{\infty} \mathbb{N} \right| = |\mathbb{N}|$$

A union involving countable many sets is called a countable union. The "countable" in "countable union" refers to the number of sets you're trying to "union-up", not the sets themselves.

Theorem 15.12.1. *The countable union of countable sets is countable.*

Instead of cobbling together more and more copies of $\mathbb{N}$, let's take a break and look at other sets. For instance there's $\mathbb{Q}$, the set of fractions. Not surely there are more things in $\mathbb{Q}$ than in $\mathbb{N}$ because $\mathbb{Q}$ contains $\mathbb{N}$ and surely the fraction $1/2$ is not in $\mathbb{N}$. Right?

WRONG!

That's because you can associating values of $\mathbb{N}$ with values in $\mathbb{Q}$ in a wrong way. You can think of $\mathbb{Q}$ as (more or less) $\mathbb{N} \times \mathbb{N}$. For instance $3/7$ corresponds to the point $(3, 7)$ in $\mathbb{N}^2$ But wait a minute ... look at the picture of $\mathbb{N} \times \mathbb{N}$. Doesn't it remind you of $\mathbb{N} \cup \mathbb{N} \cup \mathbb{N} \cup \cdots$? To be more concrete, let me show you how to associate $\mathbb{N}$ with the points in $\mathbb{N} \times \mathbb{N}$:

pic

There are some details you have to be careful about for instance What about negative fractions? The fractions also does not include something like $5/0$ (which is not defined!) However in $\mathbb{N} \times \mathbb{N}$ we do have $(5, 0)$. What about $(3, 6)$ that would correspond to the fraction $6/3$. But this is also the same as $2/1$. All these are not too difficult to overcome.

Note the technique very carefully. If you try to label the points *horizontally* like this:

you will get into trouble because you're ever going to return to start labeling the next row!!!

Your mind hurts, right?

Exercise 15.12.1. Complete the proof of $|\mathbb{Q}| = |\mathbb{N}|$ by explaining what you should do about negative fractions or about points like $(5, 0)$ or about the issue with $(6, 3)$ and $(2, 1)$. □

Now at this point, you might give up and say that $\mathbb{R}$ is probably as big as $\mathbb{N}$. Actually ... $\mathbb{R}$ is strictly bigger than $\mathbb{N}$ in size!!! Phew!!! For a moment ... we thought the whole world is nothing more than $\mathbb{N}$.

The proof that $\mathbb{R}$ is in fact bigger than $\mathbb{N}$ is important because the technique itself will be used later in a totally different context ... in the world of languages and automata.

This is so important that I need a new section!!! ...

15.13 Cantor's Diagonal Trick: Counting $\mathbb{R}$ debug: diagonal.tex

Instead of showing that $\mathbb{R}$ is bigger than $\mathbb{N}$, I'm going to show you that in fact the interval of real numbers $[0, 1)$ is already bigger than $\mathbb{N}$. The proof technique was invented by Georg Cantor, the mathematician who defined this whole new way of counting by using 1–1 and onto functions.

First of all let me just declare that I'm going to prove our goal by contradiction. I'm going to assume that $[0, 1)$ is countable. Therefore I can list the values in $[0, 1)$, say

$$x_0, x_1, x_2, x_3, \ldots$$

(The goal is to show you that I can find a value not in the above list.)

Next, I'm going to express the x_i as decimal numbers. For instance if $x_0 = 1/2$, then I'm going to write $x_0 = 0.5$ instead. Furthermore, I'm going to write the numbers with infinitely many decimal places. For instance if $x_1 = 0.5$, I'm going to write

$$x_0 = .5000000000000000\ldots$$

I'm going to give names to the decimal digits of x_0. The i–th digit is written x_{0i} So in this case,

$$\begin{aligned} x_{00} &= 5 \\ x_{01} &= 0 \\ x_{02} &= 0 \\ x_{03} &= 0 \\ x_{04} &= 0 \end{aligned}$$

etc. If x_2 is 0.14159265..., then

$$\begin{aligned} x_{10} &= 1 \\ x_{11} &= 4 \\ x_{12} &= 1 \\ x_{13} &= 5 \\ x_{14} &= 9 \end{aligned}$$

etc. So far so good ... we have a list of numbers $x_0, x_1, x_2, \ldots$ and a bunch of

digits

$$x_{00}, x_{01}, x_{02}, ...$$
$$x_{10}, x_{11}, x_{12}, ...$$
$$x_{20}, x_{21}, x_{22}, ...$$

Let me lay the digits in a grid:

	0	1	2	3	...
x_0					
x_1					
x_2					
x_3					
...					

I will fill the cell at row x_i and column j with the x_{ij}. For instance since $x_0 = 0.5000\ldots$ and $x_1 = 0.14159254\ldots$, I have the following:

	0	1	2	3	...
x_0	5	0	0	0	...
x_1	1	4	1	5	...
x_2					
x_3					
...					

Now I'm going to construct a value y that is in $[0, 1)$ that is not in the above list. I'm going to build y by specifying the decimals of y. Just like the x_i's above, y is going to look like this:

$$y = 0.y_0y_1y_2\cdots$$

where y_i is the i–digit of y to the right of the decimal point. In order for y to contradict our assumption about the countability of $[0, 1)$, I will need y not to be x_0, and not be x_1, and not be x_2, etc.

To make y not x_0, I look at the first digit of x_0 to the right of the decimal place, i.e., x_{00}. All I need to do is to choose y_0 to be different from x_{00}. For instance if $x_{00} = 5$, I can choose 0 for y_0 (of course I can also choose 1 for y_0 ... any digit that is not 5 works.) In other words I make y *not* x_0 by making

them different at the first decimal place.

So at this point y looks like

$$y = 0.0y_2y_3\cdots$$

Now suppose the second digit of x_1 to the right of the decimal point is 1. Using the same trick, I will choose y_1 to be different from $x_{11} = 1$. For instance I can choose $y_1 = 7$. At this point

$$y = 0.07y_3y_4y_5\cdots$$

Etc.

In general, for $i \geq 0$, y_{ii} to be an element in

$$\{0, 1, 2, \ldots, 9\} - \{x_{ii}\}$$

This will ensure that

$$y \neq x_i$$

Note that the form of y tells me that y is indeed in $[0, 1)$. We have found a value, y, in $[0, 1)$ which is not in the list

$$x_1, x_2, x_3, ...$$

which I assumed at the beginning is a complete list of values in $[0, 1)$.

Contradiction!!!

Actually there's a small point: What if I accidentally select $y_0 = 9$, $y_1 = 9$, $y_2 = 9$, ... This means that y looks like $0.9999\cdots$. This value is not in $[0, 1)$. This value is actually 1 and it not in $[0, 1)$. (Right? Check with your math textbooks or with your math instructors.) In the same way $0.499999\cdots$ is actually the same as 0.5. To avoid issues like these, I will just make sure that I will never choose 9 for the y_i's. Problem fixed!!!

This proof technique is called the **Cantor's diagonal trick** (or Cantor's diagonal method if you want to make it sound more respectable.)

Theorem 15.13.1. $\mathbb{R}$ *is not countable.*

Now you might think that $[0, 1)$ is much smaller than $\mathbb{R}$. But in fact ...

Exercise 15.13.1. Show that $|\mathbb{R}| = |[0, 1)|$.

I am now going to show you that there is a language (over a fixed Σ throughout this argument of course, say $\Sigma = \{a, b\}$) which is not accepted by a Turing machine, i.e., the language is not Turing–recognization (or recursively enumerable.)

I'll do this in two different ways: the first way is to show that there are more languages than Turing machines (the method is by counting) and the second way is by defining a language that is not Turing–recognizable.

First of all ... how many Turing machines are there? The number of Turing machines is actually countable.

Why?

Because each Turing machine can be encoded as a finite binary string. This means that the collection of Turing machines is a subset of

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Note that each of the Σ^n is countable (in fact finite!) Therefore Σ^* is a countable union of countable sets. This implies that Σ^* is countable.

Theorem 15.13.2. *The set of Turing machines (for a fixed Σ) is countable.*

Before we count $\textsc{Lang}_\Sigma$, the collection of languages of Σ, let's try another counting example that involves Cantor's diagonal trick.

Let's count the number of functions from $\mathbb{N}$ to $\{0, 1\}$. An element of this set is of the from

$$f : \mathbb{N} \to \{0, 1\}$$

For convenience, let me call this set X. I claim that X is not countable. By my assumption, I can list the elements in X like this:

$$f_0, f_1, f_2, \ldots$$

f_0 is of course completely determined by

$$f_0(0), f_0(1), f_0(2), f_0(3), \ldots$$

Each $f_i(j)$ is either 0 or 1. Now I'm going to build a function $g : \mathbb{N} \to \{0, 1\}$ which is not in the above list. How?

This is actually very similar to the setup in Cantor's argument above. In this case we have:

	0	1	2	3	...
f_0					
f_1					
f_2					
f_3					
...					

where the cell at row f_i and column j is filled with the value of $f_i(j)$. For instance if the table looks like this:

	0	1	2	3	...
f_0	1	0	1	1	
f_1	0	0	0	1	
f_2	1	0	1	0	
f_3	0	0	1	1	
...					

Then $f_0(0) = 1$ and $f_1(2) = 0$, etc.

Well, g is completely determined by

$$g(0), g(1), g(2), g(3), ...$$

So I need to specify $g(n)$ for $n \geq 0$. For $g(0)$, I'm going to choose a value for $g(0)$ so that g is not f_0. Well, that's easy ... I follow Cantor's diagonal trick and choose $g(0)$ to be different from $f_0(0)$. If $f_0(0)$ is 1, I will set $g(0) = 0$. If $f_0(1)$ is 0, I will set $g(0) = 1$. Likewise, I choose a value for $g(1)$ so that $g(1) \neq f_1(1)$. Etc.

In general I construct my function g such that for each $n \geq 0$,

$$g(n) = \begin{cases} 0 & \text{if } f_n(n) = 1 \\ 1 & \text{if } f_n(n) = 0 \end{cases}$$

I have constructed a function $g : \mathbb{N} \to \{0, 1\}$ such that $g \neq f_0$, $g \neq f_1$, ... Therefore the list of functions $f_0, f_1, f_2, \ldots$ cannot be complete.

Therefore X is not countable. Let's put it down as a theorem ...

Theorem 15.13.3. *The set of functions from* $\mathbb{N}$ *to* $\{0, 1\}$ *is not countable.*

Exercise 15.13.2. Is the collection of subsets of $\mathbb{N}$ countable? □

Exercise 15.13.3. The even integers are having a having a war with the odd integers. (a) Each even integer has 1 cannonball. However many ways can the even integers fire their cannonballs? If it's infinite, then is it countable? (b) What if each even integer has 5 cannonballs? (c) What if each even integer has a finite number of cannonballs? (The number can be different? (d) Countable number?

Exercise 15.13.4. Assume that all (undirected) graphs with finitely many nodes have their nodes labeled using $\mathbb{N}$. Is the collection of such graphs countable? □

15.14 Languages which are not Turing–Recognizable again debug:

not-turing-recognizable-again.tex

Now I'm going to prove that the set of languages over Σ (say $\Sigma = \{0, 1\}$) is not countable. I'll use $\Sigma = \{0, 1\}$ but the argument works in general.

First of all, recall that Σ^* is countable (see above ... remember???) Say you list the words of Σ^* as

$$w_1, w_2, w_3, w_4, w_5, \ldots$$

For instance in the case of $\Sigma = \{0, 1\}$, you can like the w_i like this:

$$\begin{aligned} w_1 &= \epsilon \\ w_2 &= 0 \\ w_3 &= 1 \\ w_4 &= 00 \\ w_5 &= 01 \\ w_6 &= 10 \\ w_7 &= 11 \\ &\ldots \end{aligned}$$

Now let's count Lang_Σ, the collection of languages over Σ. Given a language L, L contains words in Σ^*. For instance suppose

$$L = \{w_3, w_7, w_9\}$$

In that case I can construct a function $\mathbb{N} \to \{0, 1\}$ associated with L is a natural way:

$$\begin{aligned} f(3) &= 1 \\ f(7) &= 1 \\ f(9) &= 1 \end{aligned}$$

and $f(n) = 0$ for $n \neq 3, 7, 9$. In other words $f(n) = 1$ exactly when $w_n \in L$. It's clear that if I have a function $\mathbb{N} \to \{0, 1\}$ you can construct a language which is a subset of Σ^*. This associated between functions of the form $\mathbb{N} \to \{0, 1\}$ and languages of Lang_Σ is clearly a 1–1 and onto function. Therefore there must be as many languages in Lang_Σ as there are functions $\mathbb{N} \to \{0, 1\}$. I

have already shown you that the collection of functions of the form $\mathbb{N} \to \{0, 1\}$ is not countable. Therefore Lang_Σ must also be uncountable.

Theorem 15.14.1. Lang_Σ *is uncountable.*

Exercise 15.14.1. Let X be a finite set with $|X| > 1$. Is $P(X)$ countable? ($P(X)$ = powerset of X.)

The consequence is of course ...

Theorem 15.14.2. *There is a language that is not Turing–recognizable (i.e., not recursively enumerable.)*

Too bad!!! This means that there is a language that is not "computable".

The above theorem is not constructive in the sense that it does not tell you what a non–Turing–recognizable language looks like.

Let me give you an example. Although I *am* going to describe the language, I'll just warn you right away that it's not a language that is easily described using "patterns" such as

$$a^n b^n, n \geq 0$$

(like our first non-regular language) or something like

$$a^n b^n c^n, n \geq 0$$

(like our first non-context–free language).

First of all recall from above that Σ^* is countable, say the words are

$$w_1, w_2, w_3, \ldots$$

Also, note that the collection of Turing machines (over Σ) is also countable, say they are

$$M_1, M_2, M_3, \ldots$$

Notice that just like Cantor's diagonal trick, we again have a grid which is (infinitely) countable in both directions:

	w_1	w_2	w_3	w_4	$\ldots$
M_1					

M_2					
M_3					
M_4					
...					

In the cell for M_i and w_j, we put a "accept" if M_i accepts w_j. and "not accept" (not "reject"!) if M_i does not accept w_j. (Why can't you put "reject"?) To make the above easier to read, you can think of using 1 for "accept" and 0 for "not accept. For instance if the table looks like this:

	w_1	w_2	w_3	w_4	...
M_1	0	0	1	0	...
M_2	1	0	1	1	...
M_3	0	1	1	1	...
M_4	1	0	1	1	...
...					

then M_1 does not accept w_1 but accept w_3.

Following Cantor's technique, I want L not to be accepting by M_1, not accepted by M_2, etc. For L *not* be the language accepted by M_1, I will want the following to be true:

$$w_1 \in L \iff M_1 \text{ does not accept } w_1$$

In other words, if M_1 accepts w_1, then I will not put w_1 into L. However if w_1 is not accepted by M_1, I will put w_i into L. Since L has the exact opposite behavior as $L(M_1)$ in terms of accepting w_1, L cannot be $L(M_1)$. I do the same for w_2, w_3, ...

In general for $i \geq 1$, I want

$$w_i \in L \iff M_i \text{ does not accept } w_i$$

This ensures that $L \neq L(M_i)$ for all $i \geq 1$. If you want to see L in set notation, here you go:

$$L = \{w_i \mid i \geq 1 \text{ and } M_i \text{ does not accept } w_i\}$$

AHA! This means that L cannot be accepted by any of the TMs M_1, M_2, M_3, ... Since this is a complete list of all TMs, L cannot be Turing–recognizable!!!

This L is sometimes called the diagonal language. I will write

$$\textsc{Diagonal}_{\Sigma}$$

for this language. Instead of using the collection of all TMs over Σ, TM_{Σ} to form a diagonal language, you can also use the collection of all DFAs, DFA_{Σ}. So if I need to be more specific, I will write $\textsc{Diagonal}_{\text{TM}_{\Sigma}}$, $\textsc{Diagonal}_{\text{DFA}_{\Sigma}}$, etc.

Note that this is a subset of Σ^*, i.e., it is a language in $\textsc{Lang}_{\Sigma}$. (WARNING: Technically, this language depends on the ordering of both the TMs and words in Σ^*.)

15.15 Closure Laws debug: closure.tex

Theorem 15.15.1.

1. L_1, L_2 *Turing–recognizable* $\implies$ $L_1 \cup L_2$ *Turing–recognizable*
2. L_1, L_2 *Turing–recognizable* $\implies$ $L_1 \cap L_2$ *Turing–recognizable*
3. L_1, L_2 *Turing–recognizable* $\implies$ $L_1 L_2$ *Turing–recognizable*
4. L *Turing–recognizable* $\implies$ L^* *Turing–recognizable*
5. L *Turing–decidable* $\iff$ $\overline{L}$ *Turing decidable.*
6. $L, \overline{L}$ *Turing-recognizable* $\iff$ L *Turing–decidable.*

Proof. . Exercise. □

15.16 Universal Turing machine debug: universal-turing-machine.tex

The universal Turing machine, frequently denoted by M_U or U, is a Turing machine that will accept the description of a Turing machine M and an input w for M and produce the same output as M running on w. Whenever a TM M_1 runs another TM M_2, I will say that M_1 simulates M_2.

First of all, I need to tell you what I mean that the description of a Turing machine.

15.16.1 Encoding a Turing machine

Suppose you're given a Turing machine $M = (\Sigma, \Gamma, Q, q_0, q_{\text{accept}}, q_{\text{reject}}, \delta)$. First of all Γ is finite, say (for simplicity),

$$\Gamma = \{\$, \sqcup, a, b\}$$

Let me encode \$, $\sqcup$, a, b as

$$\begin{aligned}\langle \$ \rangle &= 1^1\\ \langle \sqcup \rangle &= 1^2\\ \langle a \rangle &= 1^3\\ \langle b \rangle &= 1^4\end{aligned}$$

and then encode Γ as

$$\langle \Gamma \rangle = 1^1 0 1^2 0 1^3 0 1^4$$

(It should be clear what you need to do if Γ has more symbols.) Also, say

$$\Sigma = \{a, b\}$$

then

$$\langle \Sigma \rangle = 1^3 0 1^4$$

Next say

$$Q = \{q_0, q_1 = q_{\text{accept}}, q_2 = q_{\text{reject}}, \cdots, q_{10}\}$$

Note that I'm assuming that the first state must be the initial state, the second state must be the accept state, and the third state must be the reject state. I

encode the states as follows:

$$\begin{aligned}
\langle q_0 \rangle &= 1^1 \quad \text{(initial state)} \\
\langle q_1 \rangle &= 1^2 \quad \text{(accept state)} \\
\langle q_2 \rangle &= 1^3 \quad \text{(reject state)} \\
\langle q_1 \rangle &= 1^4 \\
&\vdots = \vdots \\
\langle q_{10} \rangle &= 1^{11}
\end{aligned}$$

Note that I have "hard-coded" the initial state, accept state, and reject state. I then encode Q as

$$\langle Q \rangle = 1^1 0 1^2 0 1^3 0 \cdots 0 1^{11}$$

For the read-write head directions, $D = \{L, R, S\}$, I can choose the following encoding:

$$\begin{aligned}
\langle L \rangle &= 1^1 \\
\langle R \rangle &= 1^2 \\
\langle S \rangle &= 1^3
\end{aligned}$$

Each $\delta(q, x) = (q', y, d)$ can be easily encoded. For instance say

$$\left\langle \delta(q, x) = (q', y, D) \right\rangle = \langle q \rangle 0 \langle x \rangle 0 \langle q' \rangle 0 \langle y \rangle 0 \langle d \rangle 0$$

Clearly we can encode δ as a sequence of $\langle \delta(q, x) = (q', y, d) \rangle$ separated by 000. Then whole Turing machine M can then be encoded as

$$\langle \Sigma \rangle 00 \langle \Gamma \rangle 00 \langle Q \rangle 00 \langle \delta \rangle$$

The 00 acts as a separator between the components of M and 000 acts as a separator between the transition rules.

For instance consider the Turing machine M that replaces a and b by $\sqcup$ and

then halt.

$$\begin{aligned}
\delta(q_0, \$) &= (q_0, \$, R) \\
\delta(q_0, a) &= (q_0, \sqcup, R) \\
\delta(q_0, b) &= (q_0, \sqcup, R) \\
\delta(q_0, \sqcup) &= (q_{\text{accept}}, \sqcup, S) \\
\delta(q_{\text{accept}}, \$) &= (q_{\text{accept}}, \$, S) \\
\delta(q_{\text{accept}}, a) &= (q_{\text{accept}}, \sqcup, S) \\
\delta(q_{\text{accept}}, b) &= (q_{\text{accept}}, \sqcup, S) \\
\delta(q_{\text{accept}}, \sqcup) &= (q_{\text{accept}}, \sqcup, S) \\
\delta(q_{\text{reject}}, \$) &= (q_{\text{reject}}, \sqcup, S) \\
\delta(q_{\text{reject}}, a) &= (q_{\text{reject}}, \sqcup, S) \\
\delta(q_{\text{reject}}, b) &= (q_{\text{reject}}, \sqcup, S) \\
\delta(q_{\text{reject}}, \sqcup) &= (q_{\text{reject}}, \sqcup, S)
\end{aligned}$$

I choose the encoding

$$\begin{aligned}
\langle \$ \rangle &= 1^1 \\
\langle \sqcup \rangle &= 1^2 \\
\langle a \rangle &= 1^3 \\
\langle b \rangle &= 1^4
\end{aligned}$$

Then M is described by

$$\underbrace{(1^3 0 1^4)}_{\langle \Sigma \rangle} 00 \underbrace{(1^1 0 1^2 0 1^3 0 1^4)}_{\langle \Gamma \rangle} 00 \underbrace{(1^1 0 1^2 0 1^3)}_{\langle Q \rangle} 00 \underbrace{(1^1 0 1^1 0 1^1 0 1^1 0 1^2) 000 \cdots}_{\langle \delta \rangle}$$

(I've added parentheses to make the description readable.) Here

$$\begin{aligned}
\langle \Sigma \rangle &= 1^3 0 1^4 \\
\langle \Gamma \rangle &= 1^1 0 1^2 0 1^3 0 1^4 \\
\langle Q \rangle &= 1^1 0 1^2 0 1^3 \\
\langle \delta \rangle &= (1^1 0 1^1 0 1^1 0 1^1 0 1^2) 000 \cdots
\end{aligned}$$

where

$$\left\langle \delta(q_0, \$) = (q_0, \$, R) \right\rangle = \langle q_0 \rangle 0 \langle \$ \rangle 0 \langle q_0 \rangle 0 \langle \$ \rangle 0 \langle R \rangle$$
$$= 1^1 0 1^1 0 1^1 0 1^1 0 1^2$$

15.16.2 Hardcoding some encodings

Note that you can choose any encoding for Γ. But there are some encodings that must be fixed. For instance, the encodings for the initial state of M, the accept state and reject state must be fixed because the universal Turing machine will be using the encodings to determine how to initialize the machine to be simulated, and how to halt the machine.

15.16.3 Encoding inputs for a Turing machine

Suppose that $\Sigma = \{a, b, \}$ just like the previous section where

$$\langle a \rangle = 1^3$$
$$\langle b \rangle = 1^4$$

If $w \in \Sigma^*$ is a string which is an input for a Turing machine M, say

$$w = aaba$$

I want to talk about the encoding of w, which I will denote by $\langle w \rangle$. Note that

$$\langle w \rangle = \langle aaba \rangle = \langle a \rangle \langle a \rangle \langle b \rangle \langle a \rangle$$

is not going to work since in that case

$$\langle w \rangle = 1^3 1^3 1^4 1^3$$

So I will use 0 as separators. In other words

$$\langle w \rangle = \langle aaba \rangle = \langle a \rangle 0 \langle a \rangle 0 \langle b \rangle 0 \langle a \rangle = 1^3 0 1^3 0 1^4 0 1^3$$

15.16.4 How to setup U for execution

Note that to use the universal Turing machine U, a user must put

$$\langle M \rangle \# \langle w \rangle$$

on the input tape of U. Note that the universal Turing machine recognize the special character #. This character # is just to separate the description of the Turing machine M and the input w. (Technically speaking, instead of #, we could have used 0000 or even \$ instead of #. Anything that can uniquely tell us "end of M" would do. But it has to be fixed.)

15.16.5 How does U do its work: the big picture

During execution, U's tape would look like

$$\$\langle M\rangle\#\langle w\rangle\$x$$

where x is (basically) the instantaneous description (ID) of M when M runs on input w. Note that the description of M is never changed. Likewise the encoding of w is also never changed. In a sense the tape of U is made up of three sections. In fact in most books, U is described as a multi-tape Turing machine. Also, some books use extra tapes to keep for instance the initial state encoding, etc. Actually we don't really need to keep w, but I'm just going to keep it there anyway.

15.16.6 How does U do its work: details and encoding of ID of M

Note that the simulated ID contains the encoded characters of M and a state. For instance this might be an ID while M is running:

$$aaq_3ba$$

i.e., M is at state q_3, the tape has $aaba$, and M is about to read the third character. Of course in M, the symbols a, b, ... and q_0, q_1, ... are distinct symbols. But (in U) because we're encoding all the above into 1s the above ID

$$aaq_3ba$$

on the tape for U now is just a bunch of 1s!!!

$$1^3 1^3 1^4 1^4 1^3$$

YIKES!!! That's not going to work.

First we'll encode the above ID by add 0s to separate the encoding of the

symbols. This gives us

$$\langle a\rangle 0\langle a\rangle 0\langle q_3\rangle 0\langle b\rangle 0\langle a\rangle$$

i.e.,

$$1^3 0 1^3 0 1^4 0 1^4 0 1^3$$

Ahhh ... *now* we can see that there are 5 things in the ID.

But wait ... I still have to differentiate between the encoding of states and read-write symbols. In the encoding scheme of M, $\langle q_3\rangle = \langle b\rangle = 1^4$!!! They are encoded the same as parts of the encoded ID!!! (The reason why there's no confusion in the encoding of M is because the states and read-write symbols appear in fixed places in $\langle M\rangle$.)

What I'll do is this: since @ will never appear in the encoding of an ID of M, we can use it to tell us that what follows is a state. In other words, the above ID is encoded as

$$1^3 0 1^3 @ 1^4 0 1^4 0 1^3$$

i.e., in the encoding of an ID,

$$@1^4$$

means the fourth state in Q of M while

$$01^4$$

means the fourth character in Γ of M.

Get it?

15.16.7 How does U simulate M on w.

So say you have a TM M and you want to run M with input w. First you choose an encoding for M – remember that the encoding for the initial state, accept state, and reject state of M is fixed. With that you get the encoding of M and w. You then enter

$$\langle M\rangle\#\langle w\rangle$$

into the tape of U. (Don't forget that U understands 0, 1 and #.) The input tape looks like

$$\langle M\rangle\#\langle w\rangle$$

When you run U, it first modifies the tape to become this:

$$\$\langle M\rangle\#\langle w\rangle\$@\langle q_0\rangle 0\langle w \text{ with 0 separating symbols}\rangle$$

(Technically speaking, I don't really need to retain $\langle w \rangle$ in the second section of the tape.) In other words, U adds the encoding of the initial ID of M as the third section of the U's tape.

Suppose $w = aaba$ and I have chosen $\langle a \rangle = 1^3$ and $\langle b \rangle = 1^4$. Then the tape of U above is

$$\$\langle M \rangle \# \langle w \rangle \$ @ 1^1 0 1^3 0 1^3 0 1^4 0 1^3$$

From this part of the string:

$$\$\langle M \rangle \# \langle w \rangle \$ \underline{@ 1^1 0 1^3} 0 1^3 0 1^4 0 1^3$$

my U can see that in the simulation of M running on w, at this point, the encoded state is $1^1 = \langle q_0 \rangle$, i.e., the state of M is q_0. Furthermore M's read-write head is pointing to $\langle a \rangle = 1^3$, i.e., to a. Therefore M must execute according to the value of

$$\delta(q_0, a)$$

Suppose

$$\delta(q_0, a) = (q_4, b, R)$$

Note that

$$\left\langle \delta(q_0, a) = (q_4, b, R) \right\rangle = 1^1 0 1^3 0 1^5 0 1^4 0 1^2$$

Therefore this must be in $\langle M \rangle$.

$$\$ \underbrace{\ldots 00 \quad \overbrace{1^1 0 1^3 0 1^5 0 1^4 0 1^2}^{\langle \delta(q_0, a) = (q_4, b, R) \rangle} \quad 00 \ldots}_{\langle M \rangle} \# 1^3 0 1^3 0 1^4 0 1^3 \$ \underline{@ 1^1 0 1^3} 0 1^3 0 1^4 0 1^3$$

U will need to look for @ in the ID, then match what follows (the state and the character) with the first part of the relevant transition in $\langle M \rangle$:

$$\ldots 00 \underline{1^1 0 1^3} 0 1^5 0 1^4 0 1^2 00 \ldots \# 1^3 0 1^3 0 1^4 0 1^3 \$ @ \underline{1^1 0 1^3} 0 1^3 0 1^4 0 1^3$$

Clearly it's easy to design U so that the above match is found. Note that U has to check each transition one at a time until the right transition is found. I would need to mark the transition rule that I need to try to match. I can for instance use @ in $\langle M \rangle$ to mark the transition rule U is checking. If the marked transition rule does now what what U needs to rewrite the encoded ID (to simulate the execution of one computation step of M), then if replaces the @ with 0 and move onto the next transition rule. At some point the correct

transition rule is found:

$$\$...0@\underline{1^1 01^3}01^5 01^4 01^2 00...\#1^3 01^3 01^4 01^3 \$@\underline{1^1 01^3}01^3 01^4 01^3$$

Once a match is found, based on the rest of the string describing the transition

$$\$...0@\underline{1^1 01^3 01^5 01^4 01^2}00...\#1^3 01^3 01^4 01^3 \$@\underline{1^1 01^3}01^3 01^4 01^3$$

U modifies the encoded ID accordingly. Note that U would need to move back and force between the two @ in order to do that.

After each simulation (i.e., ID rewriting), U examines the ID and checks if the state is an accept or reject state of M. If U see an accept state, U cleans up the third section of the tape by removing the state (so the the third section of the tape of U imitates the tape of M) and then U enters its accept state. Likewise if U sees a reject state, except that U enters its reject state. Otherwise U continues simulating M.

(If I want to, I can even modify U so that before U enters the accept or reject state, U replaces its tape with the encoding of the tape of M when M halts.)

Theorem 15.16.1. *There is a Turing machine M_U (or U) such that for every TM M with input w, U accepts $\langle M\rangle\#\langle w\rangle$ iff M accepts w and U rejects $\langle M\rangle\#\langle w\rangle$ iff M rejects w. Specifically: If M rejects w by entering its reject state, then U will also enter its reject state; if M rejects w by never halting, then U also never halt while executing on the input $\langle M\rangle\#\langle w\rangle$.* □

There's something that's really really really important (but obvious), but if you're not careful you might not realize ...

Do you see that although U has finitely many states, *but* the TM that U simulates can have an arbitrary number of states. What I mean is that, say our U has 100 states. U can simulate a TM with 1000 states, and it can simulate a TM with $1,000,000$ states, and it can simulate a TM with $1,000,000,000,000$ states.

As an aside, note that for U, the substring of 0s have special meanings. Although I'm using \$ as a symbol used by U to mark beginning of tape, note that if 0^5 is not in used, then 0^5 can also play the role of \$. We can also use 0^5 intead of # too if we like.

15.16.8 Input checking

One other thing. Before U simulates M running w, it's a good idea for U to do the following:

1. Check that the input is of the form

   $$x\#y$$

   where $x, y \in \{0,1\}^*$.
2. Check the first part of input, i.e., the x in

   $$x\#y$$

   represents a valid TM M. Clearly a random string of 0s and 1s will not necessarily be a TM according to our TM encoding scheme.
3. Check that y in

   $$x\#y$$

   represents a string that the TM M can run with.

Exercise 15.16.1. (String search) Construct a TM such that when given a string of the form

$$\#x_1\#x_2\#x_3\#x_4\#\cdots\#x_n\#\#y$$

with $x_i, y \in \{0,1\}^*$, it will find the first x_i that is equal to y and mark it by changing the # in front of x_i from # to @ and then accepts. If no such x_i is found, the machine rejects the input. For instance when given this:

$$\#0010\#11000\#10101010\#11100\#\#10101010$$

The machine accepts with this on the tape:

$$\$\#0010\#11000@10101010\#11100\#\#10101010$$

If you can do the above, then you believe that the part of U that finds the relevant transition rule is doable. □

Exercise 15.16.2. (String insert) Construct a TM such then given a string of the form

$$\#x_1\#x_2\#x_3\#\cdots@x_i\#x_{i+1}\#\cdots\#x_n\#\#y@z$$

if "inserts a copy of x_i after @". For instance when given this input:

$$\#000111\#01@111\#0\#1010\#\#1010@0101$$

the TM will halt with this on the tape:

$$\$\#000111\#01@111111\#0\#1010\#\#1010@1111110101$$

If you do the above, then you believe that the part of U that changes one state encoding to another is doable. □

Exercise 15.16.3. (Moving a substring) Construct a TM that such given a string of the form

$$\#c\#x@y00z$$

where c is 1 or 11 or 111, $x, zin\{1\}^*$, and $y \in \{0,1\}^*$ that does not contain 00, then if $c = 1$, the string $@y00$ will move to the left by 1, if $c = 11$, the string $@y00$ will move to the right by 1, and if $c = 111$, the input does not change. For instance if the input is

$$\#1\#11111@1110011111$$

then the TM will change the input to this:

$$\#1\#1111@11100111111$$

and accept. If the input is

$$\#11\#11111@1110011111$$

then the TM will change the input to this:

$$\#11\#111111@111001111$$

and accept. You get the idea. If the input is not of the form above, the TM rejects.

If you can do the above, then you will believe that it's possible to move the encoding of the state in the encoding of the ID to the left or to the right or make it stay. □

Exercise 15.16.4. Construct a universal TM - mathematically. □

Exercise 15.16.5. Using the TM software provided in this class, construct a universal TM. Test it. □

Exercise 15.16.6. There are possibly many variations to the above description of U. For instance I've "hardcoded" the directions of the read-write head as

$$\langle L\rangle = 1^1$$
$$\langle R\rangle = 1^2$$
$$\langle S\rangle = 1^3$$

You can also make your more TM "flexible" and tell the user to specify the encodings of L, R, S by telling them where to put that information. For instance the encoding of a TM to be simulated can be of the form

$$\langle M\rangle = \langle\Sigma\rangle 00\langle\Gamma\rangle 00\langle Q\rangle 00(\langle L\rangle 0\langle R\rangle 0\langle S\rangle)00\langle\delta\rangle$$

Of course I have also hardcoded the fact that the first three states encoded within $\langle Q\rangle$ are the initial state, the accept state, and the reject state. Again, you can make the specification of $\langle M\rangle$ more flexible by allowing a user to choose the encoding for these states: you would have to include three places in your $\langle M\rangle$ to specify these states. For instance

$$\langle M\rangle = \langle\Sigma\rangle 00\langle\Gamma\rangle 00\langle Q\rangle 00\langle\text{initial state}\rangle 00\langle q_{\text{accept}}\rangle 00\langle q_{\text{reject}}\rangle 00\langle L\rangle 0\langle R\rangle 0\langle S\rangle 00\langle\delta\rangle$$

Note that although you're making the encoding of M more flexible, you still have to somehow hardcode some things: you have to hardcode the *position* in the encoding of M where the above are placed.

Using the TM software given in this class, build a universal TM that according to the "more flexible" specification of the universal TM above. □

Exercise 15.16.7. A TM can potentially run forever. Describe a universal TM U' that allows you to specify how long to simulate M on input w. Note that this U' will always halt: if M accepts/rejects w in n steps, then U' halts in the accept/reject state. If M does not enter its accept/reject state in n steps, then U' enters its reject state and halt. [HINT: For the input, you need the encoding of the TM, encoding of the input, and encoding of "time".] □

Exercise 15.16.8. Design a universal TM U'' that simulates two machines, running the pair in a fair manner, i.e., run one transition on one of the TM and then run one transition on the other and then repeat. If any of the two accepts, U' accepts. If both rejects, U' rejects. □

Exercise 15.16.9. Design a universal TM U''' that simulates a TM M running on w and keeps a history of its computation, i.e., it record of all the IDs. If U''' sees a two completed IDs which are the same, U''' halts in the reject state. Otherwise it halts in the same halt state of M running on w. □