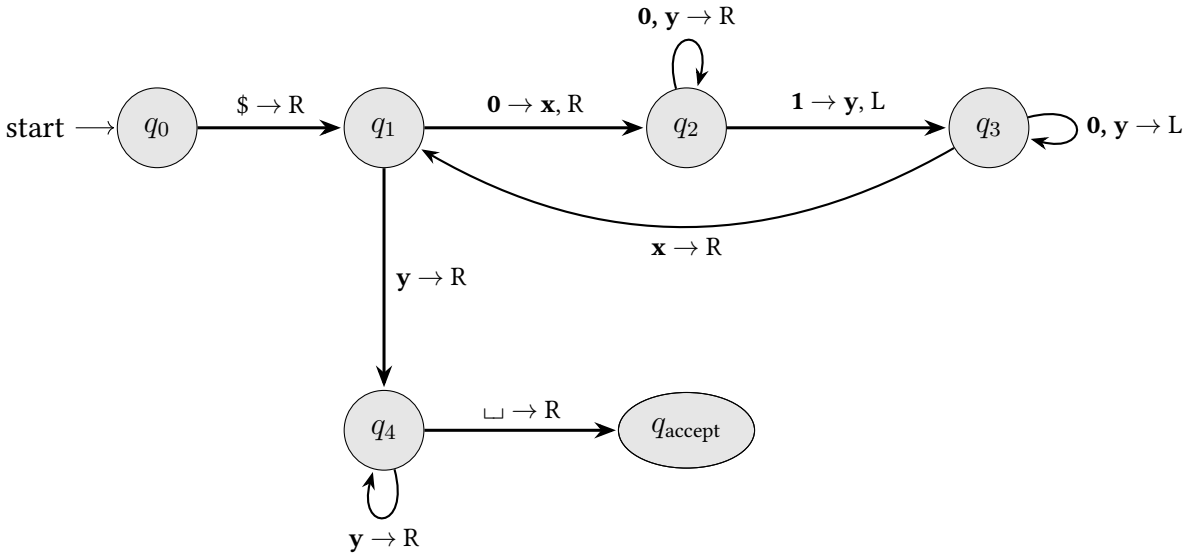
CSCC63 – Week 2

This week: More Turing Machines, the Church-Turing thesis, and an introduction to undecidability.

Turing Machines, Continued...

Let's keep talking about the TMs that we introduced last week. Specifically, recall that a TM is an automaton in which we can overwrite characters, and in which we can move either left or right on the input string:

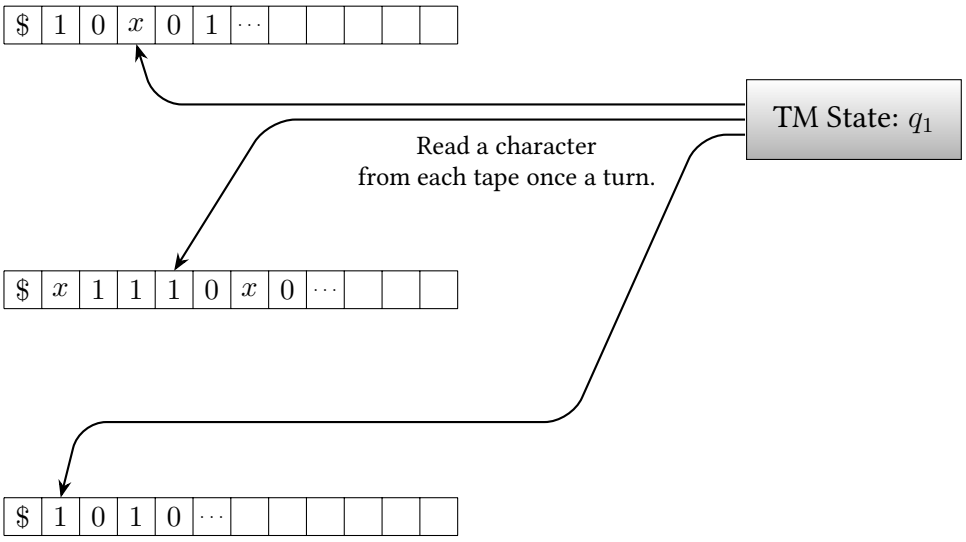


To reiterate, we stated that TMs are good for the purpose of describing algorithms because we can program them in a way that's similar to regular computers.

We also said that if there were another (feasible) way of building these machines that would even more powerful, then that would be even better. We listed the following possible approaches:

- Using more than one tape/input string/set of memory registers.
- Using non-deterministic machines.
- Allowing the TM to access arbitrary places in its memory (RAM).

Again, as a review, we described how to emulate a multitape TM using a single tape. Recall that a multitape TM is a TM with access to multiple tapes:



And that we can emulate this machine by flattening the multiple tapes onto a single one – so in the above eample, the three initial tapes

$$\$10q_1\text{x}01\llcorner\!\lrcorner$$
$$\$\text{x}11q_1 10x0\llcorner\!\lrcorner$$
$$\$q_1 1010\llcorner\!\lrcorner$$

would be flattened into

$$\#\$10q_1\text{x}01\#\$11q_1 10x0\#\$q_1 1010\llcorner\!\lrcorner.$$

Note that what we're storing are configurations – we use the memory to record where the heads are for the different tapes.

With this idea as a starting point, we used the following basic steps to emulate the multitape machines:

> Suppose we have a multitape TM $\langle M \rangle$ with n tapes, with a set Q of states, and a tape alphabet Γ. We'll write a new single-tape TM M' with a set Q' of states and a tape alphabet of Γ'. On the input w to M, the TM M' will follow these steps.
>
> 1. We'll initialize the tape of M' to $\#q_0 w \# q_0 \# q_0$, where q_0 is the start state of M and where $\#$ is a character not in Γ. We'll refer to this string, and its successors, as a flattened configuration of M.
>
> *Note that we didn't include the first "#" in the previous example. It's just there so we know where the start of the string is.*
>
> Initialize the head of M' to point to the left-hand side of this tape.
>
> 2. While M is not in a halting state (accepting or rejecting), update the flattened configuration of M as follows. To emulate one step of M:
>
> - Move the head of M' from the left-hand side of the tape to the right-hand side. Whenever we see a state for M (e.g., the q_0 in our initial string or the q_1 in our example), we move one step to the right and record that character. Note that if the next character is a "#", then we must have been at the end of the corresponding tape. We'll treat this as if M had read the blank character "$\llcorner\!\lrcorner$"

As a sanity check, remember that we "record" which character we see by using states – so we'll have a state to indicate that we've started from the left side of the string, have not yet seen an M tape head, but are in q_1, we'll have another state to indicate that we're in q_1 and that the first tape character is an "x", another state to say that we're in q_1 and that the fist and second tapes are x and 1, repsectively, and so on. All told, we'll need $\mathcal{O}(n|Q||\Gamma|^n)$ states to remember all of this. Our M' will therefore be bigger than M, but it's still theoretically possible to build.

- Once we reach the end of the tape, we know the state of M and what character each head of M sees. We now move the head of M' back to the left-hand side of its tape.

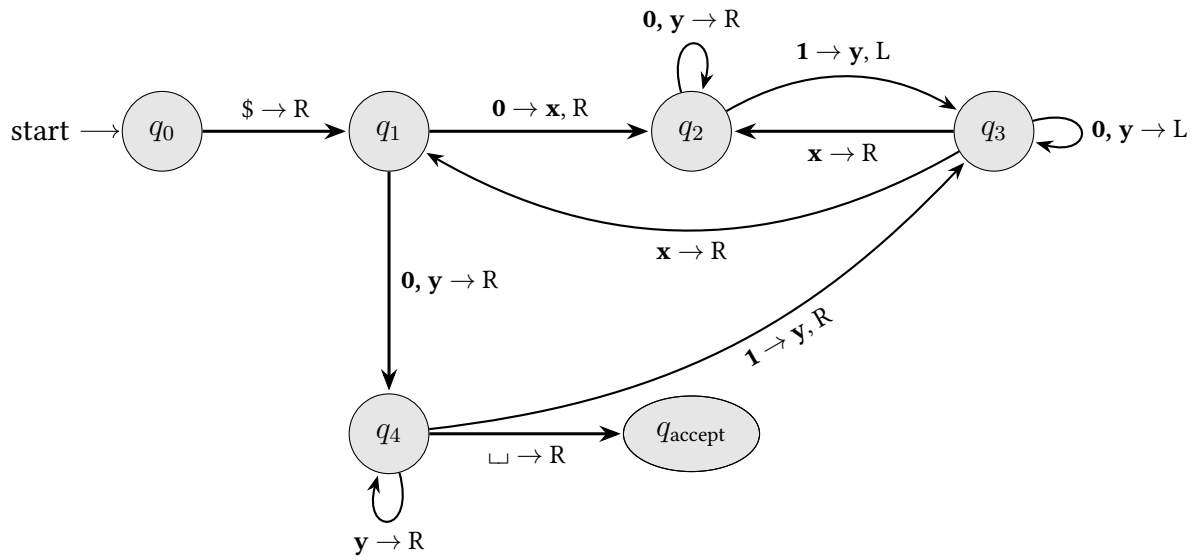
  As the head of M' moves left, it will run into the n heads of M. When it does so, we will have M' update the flattened configuration of M' as dictated by the transition function of M.

  For example, suppose that in an initial flattened configuration of $\#q_0 0101 \# q_0 \# q_0$ we wanted to update M to the flattened configuration of $\#0q_5101\#1q_5\#xq_5$. We'd start at the right-hand side of $\#q_0 0101 \# q_0 \# q_0$ and move left until we saw the final q_0. We'd update this by overwriting the q_0 with an x, moving right, and then writing q_5. We'd now have $\#q_0 0101 \# q_0 \# xq_5$, with the tape head of M' at the q_5 – we'd have updated the last of the three tapes. We'd then move left until we saw the next q_0 and update that: we'd overwrite it with a 1, move right, and see that the current character is a $\#$ (an end of string character). So we'd write a q_5 and shift the rest of the string down by one to get $\#q_0 0101 \# 1q_5 \# xq_5$. This would take some time to do, since we'd only shift the string one character at a time. But it would be doable, and we would then be able to return the tape head to the q_5 position. We'd then move the tape head to the leftmost q_0, overwrite it with a 0, move right by one, and overwrite that character with a q_5. We'd now have the string $\#0q_5101\#1q_5\#xq_5$, which is what we wanted. If we now return the head to the left-hand side of the tape, we're ready for the next step.

This approach – the idea of emulating one formalism in another – will show up many times throughout this course. We'll also see multitape TMs on occasion, since if we want to use a TM to emulate another formalism it is often easiest to start with a multitape TM: now that we know that any multitape TM can flattened into a single tape TM, we also know that if we can emulate (say) non-deterministic TM using multiple tapes, then we also know that we can do it with one tape (with some amount of slowdown).

So that said, what about non-deterministic TMS?

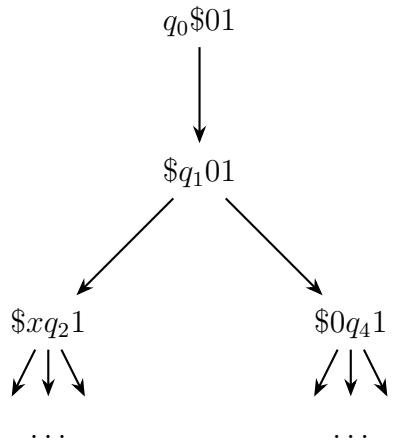
Remember that non-determinism in an automaton just means that there may be multiple arrows out of a state:



We're allowed to choose which arrow to follow as we go, and we accept if *any* choice would lead to q_{accept}. We'll refer to a non-deterministic TM as an NTM.

This means that the configuration history on this TM is more of a tree than a sequence of strings.

> If we started with the input $\$01$, the above NTM would start with the configuration $q_0\$01$, and the second would be $\$q_101$.
>
> But for the third configuration we'd have a choice of either $\$xq_21$ or $\$0q_41$. The possible computations would be:



The task of determining whether the NTM accepts is a matter of searching this configuration tree to see if there's a configuration with an accepting state.

There are a couple of ways of doing this, but the overall idea is that we perform a deterministic BFS on the configuration tree – and we'll use a multitape TM to make the job bit easier.

The idea we'll use is this: recall that when we do a BFS we keep track of the nodes we've seen but not yet explored, the node we're currently exploring, and the nodes we've already explored. In the BFS we'll do here, the nodes are labelled the configurations on the tree.

> *The labels may not be distinct, but it'll turn out that this won't matter for the particular BFS we'll run.*

So let's build a multitape TM in which the three tapes represent the three things we want to keep track of:

1. The first tape contains the configurations we've seen but not expanded.
2. The second tape contains the current configuration.
3. The third tape contains the configurations we've fully searched.

> *We won't actually need this last tape for the current search: the graph is a tree, and so we'll never need to check whether we've already seen a node. We're only putting it in for completeness sake.*

Suppose we start our search on the example tree above. We'd initialize the tapes so that the initial configuration is on the first tape. The current and expanded configurations will be set to empty tapes:

$$\begin{array}{l} \#q_0\$01\sqcup \\ \sqcup \\ \sqcup \end{array}$$

> *We'll use the "#" character as a delimiter to separate configuration strings. It'll be a character not in the tape alphabet of the NTM.*

To run a step of our BFS, we'll take the first unsearched configuration on tape one and place it on the second tape to indicate that it is the current configuration. We'll then cross off the configuration from the first tape to get:

$$\begin{array}{l} \times\times\times\times\times\sqcup \\ \#q_0\$01\sqcup \\ \sqcup \end{array}$$

> *Here I'm using the "$\times$" character to indicate a crossed-off character on the tape. We could also just erase the configuration from the first tape, but then we'd have to move all of the other configurations to the left.*

Once we've moved the current configuration to the active tape, we'll look at the transition function of the NTM to find all of the possible next moves. We'll write all of these new configurations to the first tape, after which we'll move the current configuration to the third tape and erase tape two. In our example we only have one move in step one, so we'll end up with these tapes:

$$\begin{array}{l} \times\times\times\times\#\$q_1 01\sqcup \\ \sqcup \\ \#q_0\$01\sqcup \end{array}$$

And that's one step of our search. To run our next step we'd move the second configuration to the active tape, like so:

$$\times\times\times\times\times\times\times\times\times\times\ \sqcup$$
$$\#\$q_1 01\sqcup$$
$$\#q_0\$01\sqcup$$

And after expanding this node we'd have the configurations:

$$\times\times\times\times\times\times\times\times\times\times\#\$xq_2 1\#\$0q_4 1\sqcup$$
$$\sqcup$$
$$\#q_0\$01\#\$q_1 01\sqcup$$

Note that this time we have two configurations on the first tape, but we'll still be able to find the first one by using the "$\#$" and "$\times$" characters.

And that's the idea. You can see that if there's a path from the root to an accepting configuration we'll eventually find it.

> *We've glossed over the issue of actually copying the next configurations onto the end of the string. Can you see how you would do it?*

This is somewhat high-level, but it does show that we can use a deterministic TM to simulate an NTM.

> *Just like with the multitape constructions, though, there is a time cost — the size of a tree can be exponentially large in its height, so doing a BFS of the configuration tree of an NTM can take exponentially longer than the running time of the NTM. We don't know of any faster way of doing this reduction, though… Figuring out if this slowdown is indeed necessary is one of the biggest open problems in theoretical computer science.*

Finally, we'll mention in passing the RAM model of computation. In this model, each tape cell has an address, and we can jump to any address we want during the computation. In particular, we can read memory addresses from the tape in order to figure our where to jump.

> *You can see that this basically gives us something similar to a regular computer's architecture.*

We won't describe how to implement this model in detail, but we will state that we can implement it, and that the cost is on the order of an $[f(n)]^6$ slowdown, where $f(n)$ is the length of the string.

So all told, we have several ways of implementing TMs, but they are all equivalent in terms of expressive power.

In fact, any model we can build in which:

- We allow arbitrary memory access, and
- Every step can only do a finite amount of work

Ends up having the same expressive power. This is what we call the **Church-Turing thesis:**

The Church-Turing Thesis:

Anything that we can reasonably expect to decide with an algorithm can also be decided using a TM:

Problems that can be solved using algorithms $\Leftrightarrow$ Problems that can be solved using TMs

Of course, so far we've only talked about decision problems. If we want to talk about programs that return some non-Boolean value, we just treat whatever is on the TM's tape when it halts as its return value. We'll talk more about this soon.

We've also said that some programs can loop, and that those aren't covered by our definition of an algorithm. But we've also noticed that we can get TMs to loop. What we're really saying here is that halting algorithms can be encoded by halting TMs. If we want to investigate looping algorithms we can look at looping TMs.

Virtual Machines and the Universal TM

Of course, if we're going to argue that TMs are powerful enough to encode any algorithm, then there's one specific sort of program they should be able to run: we should be able to build a TM that emulates other TMs.

After all, we can build virtual machines that take other programs as inputs and run them, and most programming languages have some sort of EVAL *function tht you can use to run strings as code. Why shouldn't we be able to do this with TMs as well?*

But how would be do this?

Well, we want to build a TM that takes other TMs (and their inputs) as arguments, and then emulates those TMs. So let's call the emulator TM as as a **universal TM**, and we'll refer to it using the variable $\mathbf{U}$[1].

Now, as we've said, the inputs to $\mathbf{U}$ will be a TM/input pair $\langle M, w \rangle$, where M is a TM description and w is an input string. Recall that the TM description $\langle M \rangle$ is a tuple that tells us:

- a set Q of states,
- an input alphabet Σ,

[1]The description we use here is adapted from a sketch in section 9.2.3 of Hopcroft and Ullman.

- an input alphabet Γ,
- a transition function δ,
- a starting state s,
- an accepting state q_A, and
- a rejecting state q_R.

Here we run into our first hurdle: different Ms may well have different tape alphabets Γ, but **U** is only allowed to use its own tape alphabet. So we'll assume that M is encoded into the tape alphabet of **U**.

> *Specifically, we'll have* **U** *work on the binary alphabet* $\{0, 1\}$. *We could encode* $\Gamma = \{x_1, x_2, \ldots, x_n\}$ *so that, e.g., the character* x_i *was denoted as* 10^i1. *Other encodings are possible as well.*

We'll make similar assumptions about the encodings of Σ and Q, as well.

To keep things simple, we'll write **U** as a multitape TM, and remember that this means that we can always flatten a multitape TM into a single tape TM with a bit of work.

We'll use three tapes for **U**:

1. A tape that stores the transition functions and the string w,
2. A tape that stores the tape of M, and
3. A tape that holds the current state of M.

Here's a sketch of how **U** works: When running **U**, we'll follow these basic steps:

1) **Initialize** the TM M: examine the input $\langle M, w\rangle$ to ensure that it is of the right format, and reject if it is not.

   > *As a general convention, an* $\langle M\rangle$ *that doesn't parse correctly is treated as a TM that halts in one step without accepting.*

   If $\langle M, w\rangle$ does have the right format, we copy $\langle \delta, w\rangle$ to the first string, w to the second, and s to the third.

2) **Run** M – while M has not yet halted:

   - If M is in q_A, halt and accept, and if M is in q_R, halt and reject.
   - Otherwise, simulate a step of M. We can read the third tape to see the current state of M, and we can count the 0s to the right of the head of the second tape to find the current character in the tape.

     > *Since we encode the states and tape characters for* M *using the alphabet* $\{0, 1\}$ *in* **U**, *we can't actually "read" a state or character in one move – instead we'll compare to the strings in* δ, *which we've stored on the first tape.*

We'll search through δ until we find a transition that matches the current configuration, and we can then update the state and tape of M to match. If there is no such transition in δ we move M to its reject state.

We'll leave further details for the reader to work out, but this is enough to justify the assertion that we can use **U** to run M. We'll generally write code for TMs using pseudocode, and the existence of **U** means that we can include in our code lines of the form:

i - 1. . . .

i . Use **U** to simulate M on w.

i + 1. . . .

or even

i - 1. . . .

i . Use **U** to simulate M on w for k steps.

i + 1. . . .

We include the reference to **U** in our pseudocode to make it clear exactly which iniversal TM we're using to run M – there are, after all, more than one universal TMs. But normally we won't really care about the details of how **U** is coded, we just care that it's possible to use one. When the specific details of how we run **U** are not necessary for our discutssion, and when the specific **U** we're using is unambiguous, we'll instead just write:

i - 1. . . .

i . Run M on w.

i + 1. . . .

With this in mind, consider the following TMs:

1) $N_H =$ "On input $\langle M, w, k\rangle$:
   1. Run M on w for k steps.
   2. If M halts within this time, *accept*.
2) $N_A =$ "On input $\langle M, w, k\rangle$:
   1. Run M on w for k steps.
   2. If M accepts within this time, *accept*.
3) $N_{\text{HALT}} =$ "On input $\langle M, w\rangle$:
   1. Run M on w.
   2. *Accept*.

4) $N_{\text{A}_{\text{TM}}}$ = "On input $\langle M, w\rangle$:

   1. Run M on w.
   2. If M accepts *accept*.

We can program all four of these TMs using **U**, and all four of these TMs accept a different language of strings:

$$\begin{aligned}
\text{H} &= L(N_H) \\
&= \big\{\langle M, w, k\rangle \;\big|\; M \text{ halts on } w \text{ within } k \text{ steps}\big\}. \\
\text{ACC} &= L(N_{ACC}) \\
&= \big\{\langle M, w, k\rangle \;\big|\; M \text{ accepts } w \text{ within } k \text{ steps}\big\}. \\
\text{HALT} &= L(N_{\text{HALT}}) \\
&= \big\{\langle M, w\rangle \;\big|\; M \text{ halts on } w\big\}. \\
\text{A}_{\text{TM}} &= L(N_{\text{A}_{\text{TM}}}) \\
&= \big\{\langle M, w, k\rangle \;\big|\; M \text{ accepts } w\big\}.
\end{aligned}$$

Note that N_H and N_{ACC} are *algorithms* – they halt on every input. So H and ACC are problems that can be decided by algorithms.

On the other hand, N_{HALT} and $N_{\text{A}_{\text{TM}}}$ are *not* algorithms – if we feed them an M that loops on its input, then both N_{HALT} and $N_{\text{A}_{\text{TM}}}$ will loop as well. It'll turn out that HALT and A_{TM} cannot be solved using algorithms. But to see why, let's give the terminology to describe it.

Our First Language Class: Decidable Languages

In this course we'll be interested in classifying languages/decision problems according to their difficulty. We'll do this by using things called *language classes*.

> *Roughly speaking, a* class *is a set of sets/languages – we'll mostly try to sidestep the problems that can occur if talk about sets of sets by stating that the elements in our sets must be objects tht we can encode as finite strings. As long as we do this, an intuitive definition of what a set is will mostly suffice for this course.*

We're interested in determining which problems can be solved by computers, and so the class of languages/decision problems that can be solved using computers is a good place to start. Using the Church-Turing thesis, we can see that this is exactly the class of languages that can be decided using TMs.

Definition: Decidability

If a decision problem can be solved using an algorithm, we refer to it as **decidable**. If it cannot, we refer to it as **undecidable**. We'll refer to the associated languages as decidable and undecidable as well[a]

[a] Decidable/undecidable languages and their decision problem counterparts, are sometimes also referred to as **recursive**/**non-recursive**. The *Hopcroft, Motwani, and Ullman* text does this, for example. You'll sometimes see the decidable languages referred to as $\mathcal{R}$ as well, and this nomenclature comes from the "recursive" notation.

Now, we'll get soon enough to languages that are not decidable. But it may be useful to start by looking at a few examples of languages that are decidable:

PRIME: $\{\langle n\rangle \mid n \in \mathbb{N} \text{ is prime}\}$

PATH: $\{\langle G, s, t\rangle \mid G \text{ is a directed graph with vertices } s \text{ and } t, \text{ and there is a path from } s \text{ to } t\}$

SQUARE: $\{\langle x, y\rangle \mid x, y \in \mathbb{N} \text{ and } y = x^2\}$

Here are a few things we can see about these languages:

- I haven't given you a TM for these languages, and I don't really need to — you already know that you can write a program for them using Python/C/whatever programming language you prefer. The Church-Turing thesis then tells us there's a TM for it out there somewhere.

  If you haven't been through CSCB63 yet, or if you took it a long time ago, you can use a breadth-first search or depth-first search to solve PATH.

- Since languages are sets, we can write them in set notation. Get in the habit of doing so, it'll make some parts of the course much easier.

- In particular, I've written each of these languages in the form

  $$\{\langle \text{instance}\rangle \mid P(\langle \text{instance}\rangle)\}$$

  where P is the predicate form of a decision problem that can be decided using TMs. This might seem a bit trivial, but you'll find it useful as well. Suppose I have a language

  $$L = \{\langle \text{instance}\rangle \mid P(\langle \text{instance}\rangle) \wedge Q(\langle \text{instance}\rangle)\}.$$

  If I want to take the compliment of this language (the set of all no-instances of L), I can write it as

  $$\overline{L} = \{\langle \text{instance}\rangle \mid \neg P(\langle \text{instance}\rangle) \vee \neg Q(\langle \text{instance}\rangle)\}.$$

If my language is defined by using a Boolean composition of many smaller decision problems, and if I need to find its compliment, I negate the Boolean formula but leave the smaller decision problems as is. As an example, if my L is

$$L = \{\langle x, y\rangle \mid x \text{ is prime and } y = x^2\},$$

then

$$\overline{L} = \{\langle x, y\rangle \mid x \text{ is not prime or } y \neq x^2\}.$$

We generally leave the domains fixed, though. If x and y are elements of $\mathbb{N}$ in L, then they are also elements of $\mathbb{N}$ in $\overline{L}$.

We'll run into more uses for this way of looking at the sets soon.

Now, applying this terminology to the languages from the previous section, we see:

- The language H is decidable, where $\mathrm{H} = \{\langle M, w, k\rangle \mid M \text{ halts on } w \text{ within } k \text{ steps}\}$.
- The language ACC is decidable, where $\mathrm{ACC} = \{\langle M, w, k\rangle \mid M \text{ accepts } w \text{ within } k \text{ steps}\}$.

It'll turn out that the other two languages are *not* decidable:

- The language HALT is not decidable, where $\mathrm{HALT} = \{\langle M, w\rangle \mid M \text{ accepts } w\}$.
- The language $\mathrm{A_{TM}}$ is not decidable, where $\mathrm{A_{TM}} = \{\langle M, w\rangle \mid M \text{ halts on } w\}$.

That is to say, if I give you a computer program M and an input w, if I do not give you a time limit, but I claim that M accepts w, there is no way for you to reliably tell if I'm telling the truth.

> *Why is this harder to solve than* ACC*?*

Recall that we actually have a TM to solve ACC, and so we know it is decidable. But the corresponding TM for $\mathrm{A_{TM}}$ can loop, and so does not encode an algorithm. So at the very least, if we want to solve the problem, we need to something different. But solving $\mathrm{A_{TM}}$ can be thought of as solving an infinite number of instances of ACC:

> Does M accept w in one step?
>
> Does M accept w in two steps?
>
> Does M accept w in three steps?
>
> ...
>
> Does M accept w in n steps?
>
> ...

Intuitively, we never run out of k to test. So if M gets caught in an infinite loop, we never reach a point when we can be sure that M will not ever accept w.

That's the idea, at least. But an idea isn't a proof. So how do we know that there is *no algorithm* that'll work to solve $\mathrm{A_{TM}}$?

Liars and Decidability

As you might guess, it's a lot harder to show that a problem is undecidable than to show that it is decidable. To show a problem is decidable we generally just give an algorithm that does the job. But to show that something is undecidable we have to show that no algorithm works — not just that we haven't yet found an algorithm, or even that no algorithm fitting some pre-made template works, but that no program — even the most opaquely written program using the most convoluted logic imaginable — could ever hope to do the job.

There are two main tools we have in this course for doing this sort of proof.

- We can do a type of proof by contradiction called **diagonalization**, or
- we can do a **reduction** from one problem to another.

There are a few other techniques, but we won't go into detail here.

Reductions are actually easier and more general (although it may not seem that way at first), and most of the proofs we'll use will be of this type. Unfortunately, reductions work under the assumption that we already know an undecidable language. To get that first undecidable language we need another technique. So we'll do reductions next week, and look at diagonalizations today.

Here's the underlying idea: we'll do a proof by contradiction by using something a lot like the ***Liar's Paradox***.

A Liar's Paradox:

Suppose I were to say to you:

> *"The sentence that I am saying to you right now is false."*

Am I telling the truth?

Well, if I'm telling the truth, then the sentence is false, so no, it can't be true.

But if I'm lying then the sentence is false, and so my statement that it's false is wrong, and so it's true…

It can't be either!

Natural language has the capacity to be inexact, and lets us do this sort of thing. But logic, where we expect every sentence to be either true or false, but not both, doesn't.

This means that we can't encode the paradox in logic.

This is a tool we can use to argue that a decision problem is not solvable. What we do is:

- Assume that a problem is solvable. If so, there's a program (and, by the Church-Turing thesis, a TM) that solves it.
- Use that program to encode the Liar's paradox.
- Check that our encoding is not consistent in order to get a contradiction.
- Use that contradiction to argue that our original assumption was wrong.

That's nice, but what sort of decision problem would let us use this tool?

If you look at the Liar's paradox you'll see that it's really all about self-reference[2]. And the languages A_{TM} and HALT using references to TMs. So if there's a TM that solves them, we'll have a TM that references another TM.

Once we have programs that run each other (and themselves), it's not too hard to find ways to make them go horribly, horribly wrong.

[2]There are variants of the Liar's paradox that are not overtly self-referential, but self-reference is such a good tool for the job, why not use it?

Great — let's do that right now!

OK, let's return to $\mathrm{A_{TM}}$:

$$\mathrm{A_{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM accepts the input } w\}$$

Why this isn't decidable: We've already given the basic steps. Let's just follow them:

- *Assume the problem is solvable.*

  OK, done — we're assuming there's a TM M_A that'll solve $\mathrm{A_{TM}}$. By our definitions, it must halt for every possible input, and give the correct answer for that input.

- *Use that program to encode a Liar's Paradox.*

  We can build programs that call other programs as functions, and we've said that we can drop programs into the input of other programs as well. We'll use these properties to write a TM D such that, given input w, D figures out what some TM (such as D itself) would do on input w, and then proceeds to output the opposite result.

  This'll be easiest to do if the input w is itself an encoded TM. If D takes as its input an encoded TM $\langle M\rangle$, we can write the program:

  D = "On input $\langle M\rangle$:

  1. Run M_A on the input $\langle M, M\rangle$.
  2. If M_A accepts, *reject*.
  3. Else, *accept*."

  This may look kind of horrible, but it works: if we have the program M_A, there's no reason we can't code this program. And since M_A halts on every input, so does this program. There's no chance for it to loop. It's an algorithm.

  The Liar's paradox worked through self-reference. What happens if we feed $\langle D\rangle$ into itself?

  We're going to argue that it's a contradiction (not surprising, since it basically flips the result of M_A). That's the next step of our process.

- *Show that we get a contradiction.*

  So we feed $\langle D\rangle$ into D as input. What happens?

  - On input $\langle D\rangle$, D (the program that's running) uses M_A to figure out in a finite amount of time what D (the input program) would do when given the input $\langle D\rangle$ (a copy of the input program). It then does the opposite.
  - So if D were to accept $\langle D\rangle$, it must reject. It can't do both, so it can't accept.
  - But if D were to reject $\langle D\rangle$, it must accept. It can't do both, so it can't reject, either…

  We have a contradiction, just like we did in the Liar's paradox. This gives us our conclusion:

- *Use the contradiction to show our original assumption was incorrect.*

  Since we can definitely build D if we have the TM M_A, and since we can't build D without getting a contradiction, it must not be possible to build M_A.

  This means that the problem that M_A was supposed to solve — the language $\mathrm{A_{TM}}$, must be unsolvable.

What we've just proven is that $\mathrm{A_{TM}}$ is undecidable.

> *What this means is that there's no computer program that can reliably determine whether other programs will accept their inputs, regardless of how much time or memory we have access to.*

So what goes wrong?

Think about what happens if you try to run D. It tries to guess its own output in order to reverse it. But then, that second-level D would just be trying to reverse another D another level down, which is itself trying to reverse yet another run of D, which is…

You just get an infinite number of runs of D, all trying to second guess each other. There's no place where you can ever stop. If you've ever seen the *Princess Bride*, this is a bit like beginning of the battle of wits, except you can't get out of it in the same way.

Here's how we do get out of the problem: our programs can accept or reject, but they can also loop forever. That's exactly what happens here.

Now, if we allow algorithms that can loop, it's easy enough to write a TM that will tell us if a TM M accepts an input w – in fact, the TM $N_{\mathrm{A_{TM}}}$ from earlier in our notes does exactly this:

$N_{\mathrm{A_{TM}}}$ = "On input $\langle M, w \rangle$:

1. Run M on w.
2. If M accepts *accept.*

You can see that the TM $N_{\mathrm{A_{TM}}}$ won't let us build the TM D. $N_{\mathrm{A_{TM}}}$ can loop, and so you can't just flip its answer. But it does accept the language $\mathrm{A_{TM}}$ in the sense that $L(M_A) = \mathrm{A_{TM}}$ — every *yes*-instance $\langle M, w \rangle$ of $\mathrm{A_{TM}}$ will be accepted by M_A, and no *no*-instance will ever be accepted.

$\mathrm{A_{TM}}$ isn't decidable, but it is the next-best thing: we can build a TM that will accept exactly the set of its yes-instances, provided that we allow it to loop on some of the no-instances. We call this sort of language **recognizable**[3].

[3] In *Hopcroft, Motwani, and Ullman* the language term is **recursively enumerable**, or $\mathcal{RE}$.

Specifically, we say a language is recognizable if there is a TM that accepts all of its yes-instances, and does that does not accept any of its no-instances. This means that every decidable language is recognizable, but not vice-versa.

There's a dual concept that applies to the complement $\overline{\mathrm{A_{TM}}}$ of $\mathrm{A_{TM}}$ — we can recognize all of its no-instances, provided that we allow the TM to loop on some of the yes-instances. We call this sort of language **co-recognizable**.

Basically, a co-recognizable language is the complement of a recognizable one. In particular, every decidable language is co-recognizable, but not vice-versa.

We'll see soon that there are languages that are neither recognizable nor co-recognizable.

But first, we've said that the decidable languages are both recognizable and co-recognizable. What about the converse?

It turns out that if a language is both recognizable and co-recognizable, then it must also be decidable. To see why, let L be a language that is both recognizable and co-recognizable. This means that there is some TM M_1 that recognizes L and another TM M_2 that recognizes $\overline{L}$.

M = "On input w:

1. For $k = 0, 1, 2, \ldots$:
2. Run M_1 on w for k steps.
3. If it accepts, *accept*.
4. Run M_2 on w for k steps.
5. If it accepts, *reject*.

You can see that this program will decide L: for any input w, either $w \in L$ and so M_1 will eventually accept, or $w \notin L$, and so M_2 must eventually accept. When either of the two machines halt we will know whether w is a yes or no instance. Notice that we can't just run M_1 or M_2 on w without constraints, since they might loop. But limiting the number of steps k we take, and slowly increasing this limit, lets us get around this problem.

This approach to running many TMs — that is, running each for a set number of steps, and slowly increasing the number of steps, is called **dovetailing**. It allows us to run different TMs on different inputs without worrying about looping. It's a very good way to build recognizers for complicated languages, so we'll see more of it later.

Now, it's easy to loop over numbers, like we do above. But we can loop over strings and TMs, as well — for example, if we have the alphabet $\Sigma = \{0, 1\}$, then we can print out the strings of Σ in shortlex order. So the order of the strings would be:

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

*You can write a program that will output exactly this sequence, and you can use this sequence in a loop. Moreover, you can see that every finite string will eventually be reached by such a loop. This sort of ordering is called an **effective enumeration** of the strings.*

*Specifically, an **enumeration** of a set S is a bijection $f : \mathbb{N} \mapsto S$, and an enumeration is **effective** if it can be compuuted by a TM.*

Practically, once we have shown, as we have above, that we can loop over a set like the strings, we don't explicitly write out the order every time we want to use it in a loop. Instead, we start our program by saying "Let $w_0, w_1, \ldots$ be an effective ordering of the strings."

I'll say without proof that you can effectively order the TM descriptions (by ordering the strings and using only those that are TM descriptions) as well, and you can make use this fact.

*On occasion, it'll be useful to make use of the index of a TM in our enumeration, and it's not hard to see that finding this index isn't too hard if we have an enumerator. So you can refer to this index as well. If you see people referring to the **Gödel number** of an object like a TM, what they're referring to is somewhat similar to an index like this.*

Now, dovetailing is very useful in conjunction with a type of machine called an **enumerator**. An enumerator is a TM that is designed to loop forever, but which has access to a special output tape. The enumerator will periodically print strings to this output tape.

Here's an example:

Let $w_0, w_1, \ldots$ be an effective ordering of the strings,
and $\langle M_0 \rangle, \langle M_1 \rangle, \ldots$ be an effective enumeration of the TMs.

M = "Ignore input:

1. For $k = 0, 1, 2, \ldots$:
2. For $i = 0$ to k:
3. For $j = 0$ to k:
4. Run M_i on w_j for k steps.
5. If it accepts, print $\langle M_i, w_j \rangle$ and continue.

You can see that this program will print exactly the strings in $\mathrm{A_{TM}}$.

You can see that in conjunction with dovetailing, this type of machine can be very powerful — we can even show that a language is recognizable iff there is some enumerator for it.

This was just an example, but we'll work on a proof later.

OK, so that's almost all we need to see for today. Let's just finish off by looking at how we'll go on to prove other languages are undecidable (or even unrecognizable).

We're not going to have to use diagonalization over and over — we might see it again once or twice, but we really only need it for that first step. What we're going to do instead is this:

- Pretend we have access to a TM (or a programming library, or a black box) that we can call on to solve L for us.

- Use that TM to write a program that would solve another problem we know is undecidable, such as A_{TM}.

Let's use this approach to argue that another language is undecidable:

$$\text{HALT} = \{\langle M, w\rangle \mid M \text{ halts on input } w\}$$

It's easy to see that HALT is recognizable — just run M on w, and accept once it halts.

But showing that it isn't decidable is also not too bad once we know that A_{TM} isn't decidable — suppose that we had a TM M_H that could decide whether $\langle M, w\rangle \in \text{HALT}$. Then we could write the following program:

M' = "On input $\langle M, w\rangle$:

1. Run M_H on $\langle M, w\rangle$.
2. If it rejects:
3. *Reject.*
4. Else:
5. Run M on w.
6. If it accepts, *accept*, otherwise *reject.*

This program would decide A_{TM}, and we know that's impossible. So we know there's no such TM M_H. HALT is not decidable.

Question: *Do you think that* HALT *is co-recognizable?*

And we'll mostly leave off with that idea today. Next time, we'll refine the argument we made for HALT and use it to start talking about *mapping reductions.*

The Take-Away from this Lesson:

- We've described a couple of alternative TM formalisms, including NTMs.
- We've described the Church-Turing thesis.
- We've defined decidability, recognizability, and co-recognizability.
- We have described two languages that are recognizable but not decidable: ACC_{TM} and HALT.
- We have introduced the concepts of dovetailing and enumerators.
- We have started describing how we will prove undecidability in the rest of the course.

Glossary:

The Church-Turing thesis, Co-Recognizable, Decidable, Effective Enumeration, Enumerator, Non-deterministic TM (NTM), Recognizable, Undecidable

***Languages*:**

ACC,A_{TM}, H, HALT