Notes:
- Attend tutorials: one question on the midterm and final will come from tutorials

CSCC63 – Week 1

This week: Course introduction and definitions.

Welcome to CSCC63!

Let's start with the administrative details. I'll be your teacher this term — my name is Eric Corlett, and you can find my email at eric.corlett@utoronto.ca.

My office hours will be held on Tuesdays from 3:00 to 4:30 in IC402. The TAs will also be holding office hours. Please come in if you have any questions about the course material. You can also send me an email if you need.

Course information can be found on Quercus – in particular, we'll be using it for the course notes. The course syllabus can also be found there. The assignments will be submitted using Crowdmark.

Let's take a few minutes to go over what's in the syllabus.

If you can do so at this moment, please go on to Quercus
and have a look at the syllabus.

With that done, we'll spend the first lecture giving a course overview and discussing background material.

What is this course about?

This course is here to get you started in thinking about the *limits of computation*: what sort of problems can we reasonably compute using algorithms, and what problems will we never be able to fully answer? And if we can compute the answer to a problem, how efficiently can we do so?

There are a number of obvious candidates for questions we can't answer using computers and algorithms — e.g.,

What is your favourite colour?

What is the meaning of life?

Is there a god?

> *We won't even try to address these questions. Instead, we'll look for clear logical questions whose answers we still cannot find.*

Of course, we're designing more powerful computers all the time. So a statement about the limitations of your laptop today may not apply in a few years. In order to give limits that won't change, we have to talk about the limitations of logic itself.

We'll divide our questions about logic (and the course) between two main topics:

- **Computability**: What questions can logic ever solve?
- **Complexity**: What questions can logic solve efficiently?

Roughly speaking, the first half of the course will deal with the first topic, and the second half will deal with the second.

Some Terminology

In both topics, though, we'll need to have some way of talking about the problems we are likely to have to solve. A lot of what we do will relate to what we call **decision problems**. A decision problem is a problem whose answer is either *Yes* or *No* (alternatively, *True* or *False*). We'll start by talking about these problems, and then extend our discussion to other types of problems later.

> **Some sample problems:**

- Given integers x and y, what is $x^2 + y^3$?

  (not a decision problem)
- Given integers x, y, and z, is $z = x^2 + y^3$?

  (decision problem)
- Given integers x and z, is there an integer y such that $z = x^2 + y^3$?

  (decision problem)

So you can see that we can ask many similar problems, some of which are decision problems, and some of which are not.

> *Can you see the difference between the first two problems and the last? It's an idea that we'll return to soon.*

Of course, not all questions that we'll ask will be about numbers — we can also ask questions about graphs, geometric objects, or even other computer programs!

If we're dealing with a problem, decision or not, we'll want to talk about different problem inputs. Now, we don't like to use the term "inputs" when referring to problems — an input is something that we give to a program, and we want to distinguish programs from the problems that they solve. So we'll refer to an "input" for a problem as an **instance**. If a problem is a decision problem, the inputs that should give a *Yes* answer are *yes-instances.* Similarly, the inputs that should give an answer of *No* are *no-instances.*

Inputs to problems concerining computability: encodeable instances

We've said that we can ask questions about lots of different objects such as graphs or computer programs. These objects will then be part of our problem instances.

We'll need to make one small limitation to the objects that we ask questions about, though — we have to be able to encode them into a form that we can put on a computer. So we won't ask questions about objects that have no reasonable encoding.

Now, most objects we care about do have reasonable encodings, but things like real numbers (as opposed to their floating-point approximations) and arbitrary sets of natural numbers would take an infinite amount of memory, and so we won't work with them here.

Something to Think About:

We've just described two decision problems:

- Given integers x, y, and z, is $z = x^2 + y^3$?

and

- Given integers x and z, is there an integer y such that $z = x^2 + y^3$?

We can equally well describe these problems using set notation – specifically, by writing the sets that encode the *yes*-instances of the problems:

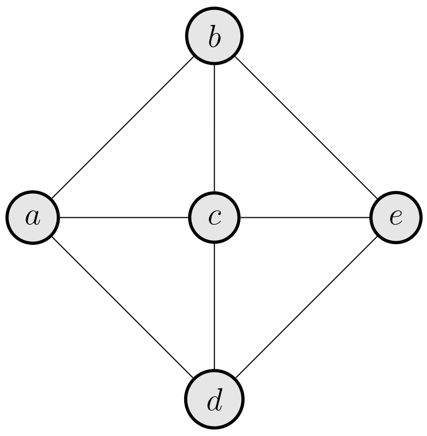
- $S_1 = \left\{(x, y, z) \in \mathbb{Z}^3 \,\middle|\, z = x^2 + y^3\right\}$.

and

- $S_2 = \left\{(x, y) \in \mathbb{Z}^2 \,\middle|\, \exists z \in \mathbb{Z}, z = x^2 + y^3\right\}$.

Many of the ideas we work with in this term will be easier to work with if you're used to using this sort of set terminology. So make sure you practice.

As an example of something we can **encode** *into a computer, if we wanted to work with the graph*



we'd have to encode it in a machine-readable format, say through an adjacency matrix like

	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	0	1
c	1	1	0	1	1
d	1	0	1	0	1
e	0	1	1	1	0

We refer to the graph itself as G, but we distinguish the **encoded format** *for the input by using angle brackets, like so: $\langle G \rangle$.* **Be careful about this distinction!**

Now, one nice thing about decision problems over machine-readable instances is that they correspond nicely with the idea of *languages* from CSCB36. Recall from CSCB36 that an **alphabet** Σ is a finite set of characters. If we're dealing with computer inputs we often use the alphabet $\Sigma = \{0, 1\}$.

Once we have an alphabet fixed we can talk about the **languages** over Σ^* — that is, the sets of finite strings using the characters from Σ.

Since languages are a type of set, we'll use set notation to describe our languages.

To repeat our earlier thought, there is a link between languages and decision problems: the language of a decision problem is its set of yes-instances, while the decision problem for a language would be: "Is this input part of our language?".

So we're going to see that there's a lot of overlap between the formal languages from CSCB36 and the decision problems we deal with now.

Before we move one we'll have one more definition — we need to make sure we're clear about what we mean when we talk about *algorithms*. Intuitively, we know what an algorithm is — it's a sequence of logical (mechanical) steps we can use, or get a computer to use, to solve a problem. The extra bit that we need to include today, though, is that we're going to require that our algorithms halt on all possible inputs.

Definition: Algorithm

An *algorithm* to solve a problem is a logical sequence of steps which will, for any problem instance, terminate in a finite amount of time and give the solution for that instance.

Now, this limitation means that some programs — such as operating systems — don't qualify as algorithms. On the other hand, operating systems are still not meant to loop forever on any individual problem — they wait for input, and when they get it, they call subroutines to deal with that input. Those subroutines do have to halt.

We can define models of computation that don't halt as they solve problems, as well. But if they don't halt we won't be able to be certain about the result of the computation – and so naturally the resulting models are of limited use. But they can have some uses, and we'll investigate one such formalism in assignment 1.

Algorithms as Mathematical Objects

> *OK, so we have a bunch of terms to use. But how does that help us talk about the limitations of logic and computing?*

Well, the first step to getting to the limitations of logic is to give terminology that we can use to talk about it. The next step is to figure out how we can encode a general algorithm as a mathematical object.

> *One of the problems we have to consider is: if I argue that every algorithm can be encoded as a particular type of object, and I use that description to argue that there is something that an algorithm cannot do, then you could call my result into question by claiming that some algorithms can't be encoded using that description — that you need a more powerful formalism. This is a difficult issue to get around, since it involves finding a model that can encode all algorithms, regardless of how imaginative they might be.*

In the early 20^{th} century, two mathematicians named David Hilbert and Wilhelm Ackermann raised the question of how to build an algorithm to determine whether one set of statements in first-order logic implies another [1]. We now know that no such algorithm exists.

But proving that no algorithm can solve this problem was itself problematic — the only way that we could talk about algorithms was to describe specific algorithms, which isn't enough to prove nonexistence.

In order to better understand algorithms, a number of mathematicians built formalisms for describing algorithms – notably, Alonzo Church built something called the *lambda calculus* and Alan Turing built a type of automaton that had arbitrary memory.

In the end, it turned out that while we could build each of these formalisms in the real world, we couldn't build anything with more computational power…in particular, all of these formalisms can emulate each other. What's more, while each of these formalisms was meant to describe what a human could reasonably do with pencil and paper, we can also use them to emulate computers, as well. So we argue that they provide a framework for describing the algorithms that computers are capable of running.

[1] There were many similar questions, all regarding different problems. This is just a convenient one.

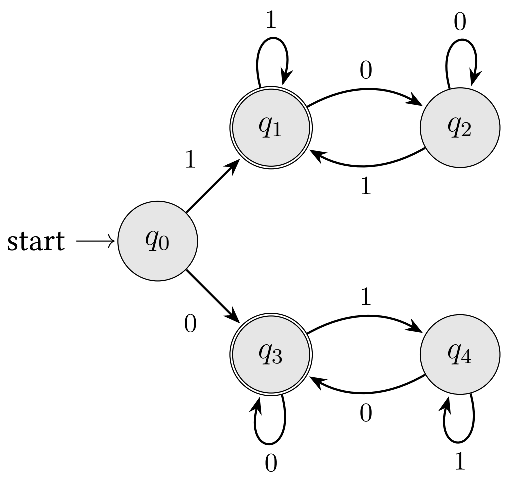
In this course we'll use Alan Turing's formalism — a type of automaton called a *Turing Machine.*

Let's just review a couple of simple automata before we describe the full Turing machines:

In CSCB36 we saw DFSAs - these were basically a type of flow chart we could use to determine membership in languages. They consisted of states and arrows, and had no memory.

As an example, here's a DFSA to determine membership in the language

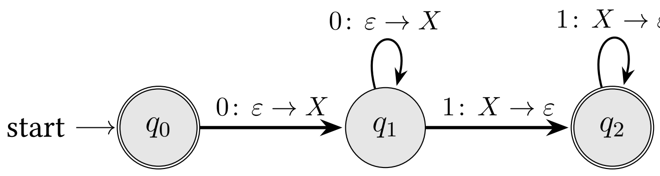
$$L = \left\{x \in \{0,1\}^* \,\middle|\, x \text{ begins and ends in the same character}\right\} :$$



You feed your string x into the machine, follow the arrows at every character, and accept iff you finish in an accepting (double circle) state.

DFSAs can accept a lot of things, but since they have no memory they can't recognize anything that requires them to count, for example the language $L = \left\{0^n1^n \,\middle|\, n \in \mathbb{N}\right\}$.

In CSCB36 we described a machine called a PDA that can recognize this language: We used the same basic machinery but gave it a stack for memory:


start
q_0
q_1
q_2
$0\colon\ \varepsilon \to X$
$0\colon\ \varepsilon \to X$
$1\colon\ X \to \varepsilon$
$1\colon\ X \to \varepsilon$

The stack memory allows our machine to count, but since the memory is a stack, it has to be accessed in order — so it can't interleave patterns. It also is limited in its calculations to the size of the input string. So we can't recognize, for example, the language $L = \left\{w\#w \,\middle|\, w \in \{0,1\}^*\right\}$.

As a sanity check: the strings in L consist of a repeated binary string w, separated by a "#" character.

A **Turing Machine** (we'll call it a TM to shorten it) is a much more powerful variant of these automata. It allows us to use arbitrarily large computing times, and it gives us access to unrestricted memory. This is achieved using two changes from the previous designs:

- In order to get unrestricted computing time, the head on the tape (the position of the character in question) can move either right or left upon reading a character. The direction of movement is determined by the arrow in the machine.

  We treat the string as having an unlimited number of blank ␣ characters to its right.

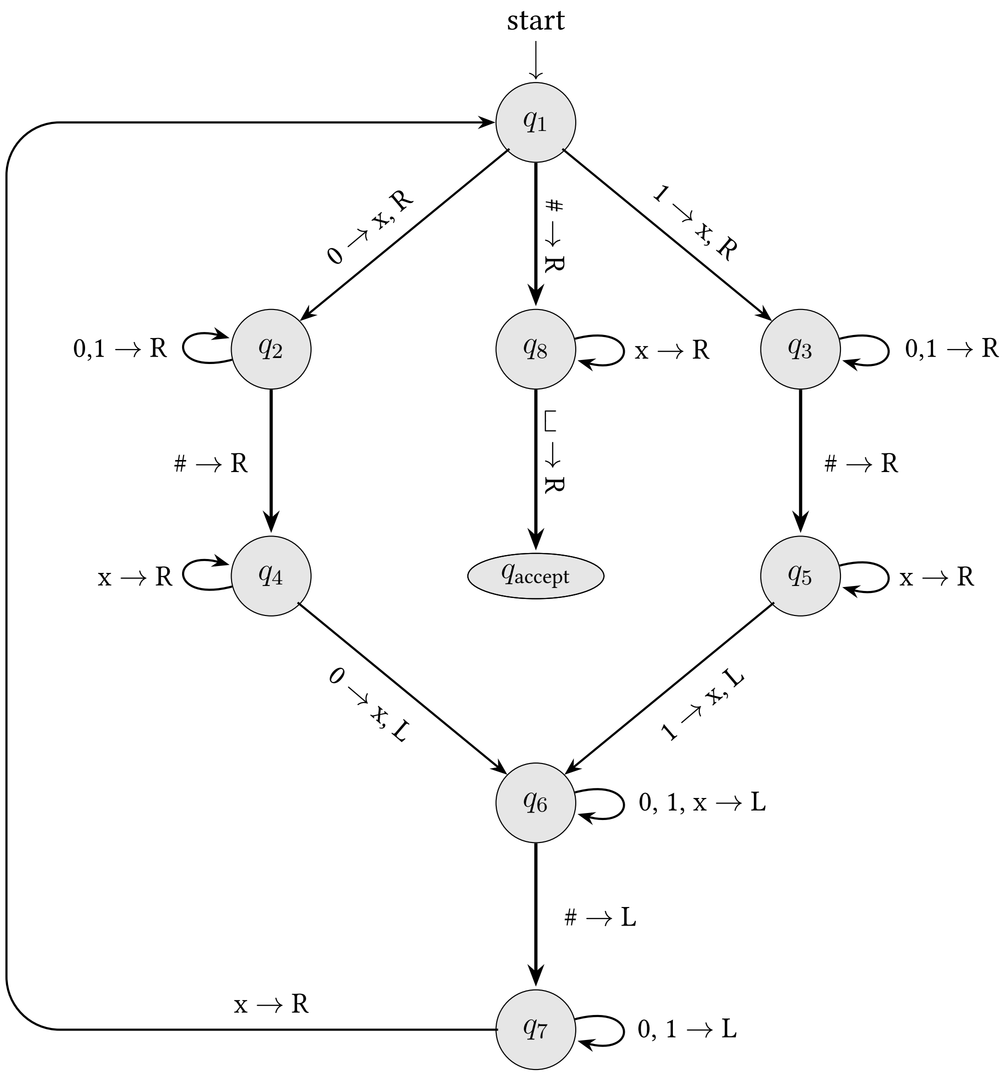
- In order to give the machine unrestricted memory, we allow it to write on the tape. As before, it is the arrow on the diagram that tells us what to write.

So how could we use this type of machine to recognize, say, $L = \left\{w\#w \,\middle|\, w \in \{0,1\}^*\right\}$?

Well, how would we normally try to recognize this language?

> *One easy way to do it would be to compare the first character of each string, then the second, and so on…*

This is basically what we'll do. So here's a TM that follows this approach[2]:



If this seems confusing, don't panic too much — this is just an example, we'll formalize it soon. But right now, let's play with it a bit and see how it all works…

[2]Taken from example 3.9 in the text.

Let's see what happens if we feed the string 010#011 into this machine.

At the beginning, it'll just be like any other automaton we've seen before: we start at the state q_1, and we'll be looking at the character on the left side of the string.

We denote this position in this way:

$$q_1 010\#011$$

Now, the first thing the machine will do is read the first character — the first 0.

When it does, it'll follow the $0 \to$ x, R arrow and move to the state q_2 — so far, that's just like every other automaton you've seen.

But there's a bit more that it'll do — you can see there's a "x, R" on the arrow. Those symbols mean that we overwrite the 0 with an x and move right on the string. So now we'll be in the position

$$x q_2 10\#011$$

Next, we see the character 1. So we follow the loop $0, 1 \to$ R. This arrow doesn't tell us write anything, just move to the right. So now our position is

$$x1 q_2 0\#011$$

Let's continue. If we keep following the arrows, the next couple of positions will be:

$$x10 q_2 \#011$$
$$x10\# q_4 011$$

Now, we're in q_4 and we're seeing a 0. So we need to follow the $0 \to$ x, L arrow. This time we need to write an x again, but instead of an R we see an L after the arrow — this means that we move left on the string:

$$x10 q_6 \# x11$$

And that's basically what we'll keep doing. If we let this TM keep running through the string this way, it'll eventually reach the position of

$$xxx\#xx q_4 1$$

.

At this point, there'll be no arrow we can follow, so the TM will reject.

Alternatively, the TM enters a special reject state, and then rejects.

And that's the basic idea.

Let's also see what happens if we feed $1\#1$ into the machine — now we see the arrangements:

$$\begin{gathered} q_1 1\#1 \\ x q_3 \#1 \\ x\# q_5 1 \\ x q_6 \# x \\ q_7 x \# x \\ x q_1 \# x \\ x \# q_8 x \\ x \# x q_8 \sqcup \\ x \# x \sqcup q_{\text{accept}} \sqcup \end{gathered}$$

This time the TM enters the q_{accept} state, in which case we say it accepts.

And that's how a TM works — if we want a formal definition, it's a lot like the definitions we used for DFSAs and PDAs. That is:

Definition: Turing Machine

A TM M is a machine described by a tuple

$$\langle M \rangle = (Q, \Sigma, \Gamma, \delta, s, q_A, q_R),$$

where:

- Q is a set of states — the nodes in the above diagram.
- Σ is the input alphabet — this is the set of available input characters. In the above diagram $\Sigma = \{0, 1, \#\}$. The blank symbol $\sqcup$ is never in Σ.
- Γ is the tape alphabet — this is the set of available input characters, plus any other characters we might see in the course of our computation. In the above diagram $\Gamma = \{0, 1, \#, \text{x}, \sqcup\}$. Note that $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma$.
- δ is the transition function — the arrows on the diagram. Formally, we denote it as a function

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

  but it's probably easier to think of it as the set of arrows — the formal definition is just the encoding.
- s is the start state — it's the state q_1 in the above TM.
- q_A is the accept state — if we ever enter this state we halt and accept. In the diagram above it's q_{accept}.
- q_R is the reject state — if we ever enter this state we halt and reject.

Some extra points:

- *As we saw in the example above, if we ever reach a point where there's no arrow out of our current state, we immediately go into an implicit reject state, reject, and halt.*
- *If we're on the left side of the tape and try to move left, we just stay in the same place.*
- *We'll extend the terminology we used for DFSAs and PDAs and write $L(M)$ to indicate the language of strings accepted by a TM M.*

And so that's what a TM is. We'll just give one more definition — the positions that we wrote out in our first example are generally called **configurations**.

Now, we'll use TMs for a number of things, but we won't spend too much time on the automata themselves: we like them for two reasons:

1. We can basically program them the way we do regular computers.
2. Their configurations are fairly well-behaved, and so we can do math on them. This will be what allows us to relate TMs (and computability results on TMs) to non-TM objects later on.

We'll look more at the second idea later, but for now, let's finish off by building a TM for ourselves: try, as an exercise, to build a TM that recognizes the language

$$L = \left\{1^{2^n} \,\middle|\, n \in \mathbb{N}\right\}.$$

A Few Conventions

Here's a reference for a few notational conventions that I've has questions about in the past:

- The set $\mathbb{N}$ of natural numbers will include 0.

  If we want to denote the positive integers starting at 1, we'll use the term $\mathbb{Z}_+$ (similarly, the negative integers will be referred to as $\mathbb{Z}_-$).

- If $a, b, c \in \mathbb{Z}$, then $a \equiv b \pmod{c}$ iff c divides $b - a$.

  So $7 \equiv 2 \pmod{5}$ and $-1 \equiv 3 \pmod{4}$.

- If A and B are sets then $A \setminus B$ is the set difference between A and B: i.e., the set $\{x \in A \,|\, x \notin B\}$.

  The notation $A \oplus B$ will refer to the symmetric difference between A and B: i.e., $A \oplus B = (A \setminus B) \cup (B \setminus A)$.

Further notation will be added as necessary.

The Take-Away from this Lesson:

- We've gone over the course syllabus and some course policies.
- We've given an outline of the course topics.
- We've given a small description of some of the terms we'll be using in this course.
- We've seen an example of a *Turing Machine*, as well as a formal definition.

Glossary:

Algorithm, Alphabet, Configuration, Decision problem, Encoding, Instance, Language, Turing Machine