Lecture 12: Turing Machines

built on 2021/02/11 at 09:30:02

1 Towards A Universal Notion of What Is Computable

This lecture will be in part philosophical and in part technical. Let's start with the philosophical part:

How can we ***mathematically*** *define computation? How about an algorithm?*

We probably have an intuitive understanding of what computation entails and what an algorithm means. But in this present case, we want precision: we want to be able to define it mathematically.

In the most basic form (but good enough for our purposes), a computational problem is simply a function from input to output. Hence, given the right alphabet Σ, we can define a computational problem to be a function

$$f : \Sigma^* \to \Sigma^*.$$

Should we place any restriction on what kind of functions? Are there functions that can be described but cannot be computed?

1.1 How to define computation?

We have to start with something: Say a computation is the execution of a Python program on some input. Why not, right? We can also define an algorithm to be a function written in Python (of course, allowing access to as much memory as it needs). Does this make you happy?

On the surface, this would be okay except: what would you say to C programmers? and to Java programmers? and to Haskell programmers? Maybe you'd say this:

Claim: What's solvable in Python == What's solvable in C (or your favorite language _fill in the blank_)

And this must be true because (1) you can write a Python interpreter in C; and (2) you can write a C interpreter in Python.

At the end of the day, the choice of language doesn't make much of a difference here: they're all the same. However, no matter what (high-level, complex) language you pick, there is a problem: the precise specification of that language is insanely complex—no one knows every deep dark corner of Python, Java, or C.

A good (mathematical) model should be simple! It would be neat to have a small "totally minimal" programming language that (1) can simulate all these languages, and (2) is simple enough for rigorous reasoning mathematically.

As it turns out, there are a few such languages. We'll look at Turing machines today and some others next time.

2 Turing Machines

We are now used to the idea of a machine keeping a state and as it reads the input, the state changes according to a predefined set of rules (state transition). In the case of PDAs, we gave it some extra memory (in fact infinite, meaning as much it needs) but that can only be accessed in a last-in, first-out manner. To create a more sophisticated model—one that will hopefully be as powerful as Python/Java—we'll equip it, again, with infinite memory. But this time, read/write is not restricted to just one end.

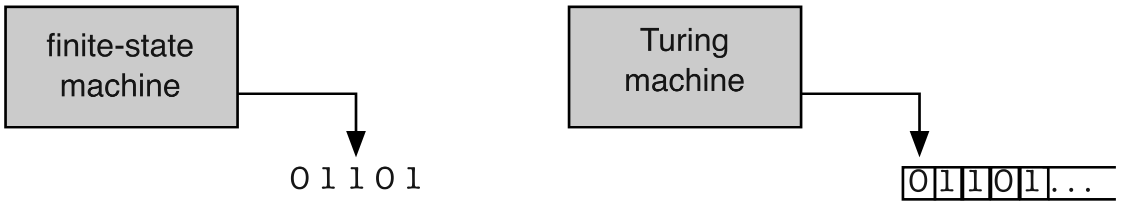
2.1 At A Glance

First proposed by Alan Turing in 1936, the so-called Turing machine is similar to a finite automaton in that it maintains a state but is equipped with a unlimited (i.e., as much as you want to use) and unrestricted-access memory.

The memory is modeled as an infinite-length tape, with a *tape head* for reading and writing symbols. The tape head sees one square at a time but can move around on the tape. Each square has a capacity of one symbol.

Warning: there are many other variants.

The tape initially contains the input, which stretches from the beginning of the tape (leftmost) out as far as is needed. Beyond that, it is blank and each square is marked with the symbol ␣.



2.2 More Formally

Definition 2.1 (Turing Machine) *A* Turing machine *is a 7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$*, where* Q, Σ, Γ *are all finite sets and*

- Q*: a set of states*
- Σ*: the input alphabet* ($\sqcup \notin \Sigma$)
- Γ*: the tape alphabet, where* $\sqcup \in \Gamma$ *and* $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$*: the transition function ("the instructions")*
- $q_0 \in Q$ *is the starting state*
- $q_{accept} \in Q$*: the accepting state*
- $q_{reject} \in Q$*: the rejecting state* ($q_{accept} \neq q_{reject}$)

Rules of Computation:

- The control starts in state q_0 with the head in the leftmost square.
- Initially, the tape has the input $x \in \Sigma^*$, followed by infinite ␣s.
- If the current state is q and the head is on a symbol $s \in \Gamma$, the following things will happen:
  - $\delta(q, s)$ gives q', s', d
  - q' tells us the next state to be in
  - s' tells us to overwrite the current square on the tape with s' (replacing s)
  - d tells us to move the head either left ($\leftarrow$) or right ($\rightarrow$).
- Many variants: OK to assume trying to move left at the leftmost square doesn't go anywhere; you stay put.
- Keep working until you reach q_{accept} or q_{reject}. At that point, the machine stops and returns "accept" or "reject."

Remarks: This is the right time to make a few important remarks about TM behaviors, which differ in significant ways from FSMs:

- First, the machine may never finish: we say that it *loops*.
- The machine may accept or reject before it reads all the input.
- The machine may accept or reject long after it reads all the input.

2.3 Decidable Languages

We are going to use the verb 'decide' in a specific way:

Definition 2.2 *A language* $L \in \Sigma^*$ *is* decidable *if there is a Turing machine M such that*

- *M halts on every input* $x \in \Sigma^*$*; and*

- *M accepts every $w \in L$ and rejects every $w \notin L$.*

This M is called a *decider* as it *decides* the language L.

Notice that an important characteristic of a decider is that it doesn't loop: it is decisive—given a string, it says yes or no.

2.4 What's Computable?

A function $f : \Sigma^* \to \{0, 1\}$ is *computable* if the language $L_f = \{x \in \Sigma^* \mid f(x) = 1\}$ is decidable.

Let's also extend this to computational problems that don't just output True (1) or False (0).

A function $f : \Sigma^* \to (\Gamma \backslash \{\sqcup\})^*$ is computable if there is a Turing machine M that (1) halts on every possible input $x \in \Sigma^*$ and (2) the resulting tape contains $f(x)$ followed by $\sqcup$s.

2.5 An Example

Example 2.3 *Consider the language $B = \{0^{2^n} \mid n \geqslant 0\}$. We'll show that B is Turing decidable. The proof is by construction: we'll give a TM M_2 that decides B:*

On input string w:

1. *Sweep left to right across the tape, replacing every other 0 with $\sqcup$ (crossing them off).*
2. *If in stage 1 the tape contained a single* 0 *(how?), accept.*
3. *If in stage 1 the tape contained more than a single* 0 *and the number of* 0*s was odd (how?), reject.*
4. *Return the head to the left-hand end of the tape (how?)*
5. *Go to stage 1.*

The nitty-gritty details are tedious; I refer you to Sipser. Suffices to say, given a "medium-level" description such as above, it can be translated into a legit low-level TM (if you have enough time to waste).

2.6 Unary Encoding

How would you represent a number k on a TM? One easy way is to use the unary representation: that is, to represent $k \geqslant 0$, you write k many 1s on the tape.

Hence, for a function $f : \mathbb{N} \to \mathbb{N}$, a TM that computes this function would

- takes as input 1^n on the tape; and
- produces as output $1^{f(n)}$ on the tape.

Example 2.4 *The function $f(n) = n + 1$ is Turing computable.*

Well, we walk to the end of the input and write down a 1.

Example 2.5 (TM can add) *The function $g(m, n) = m + n$ is Turing computable.*

First, let's talk about encoding the input m and n. An easy way is to put $1^m\#1^n$. Once this is done, do you see how to do it?

Example 2.6 (TM can multiply) *The function $h(m, n) = m \times n$ is Turing computable.*

Assuming the input is stored as $1^m\#1^n$, the basic idea is to keep decrementing m while adding a copy of n to the end. Notice that this will not keep the output at the beginning of the tape, but this can be fixed by copying the data to the beginning after you're done.

You probably have gathered by now that designing Turing machines with diagrams etc. tend to be tedious. We'll start exploring classes of functions that are flexible and robust, and show that they can be mapped to TM. Just as you wouldn't write a program directly in Assembly, you probably don't wanna express an algorithm directly as a TM.

3 Practice Exercise

Watching the following video detailing how TM works at the low level.

https://www.youtube.com/watch?v=gJQTFhkhwPA

After that, provide a low-level description of the TM (like in the video) for the following tasks:

1. Move right until the first ␣ is encountered.
2. Sweep left to right across the tape, replacing every other 0 with ␣ (crossing them off).