

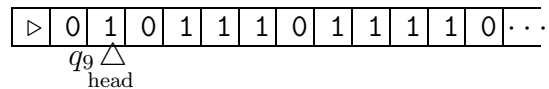
Theoretical CS: Turing Machines Explained

Mathcamp 2006, Week 2

September 15, 2006

1 A computer with memory

A Turing machine is a program along with memory. We let the Turing machine have infinite memory in the form of a tape of cells. Each cell holds one character. The machine doesn't have any way of going straight to any particular cell, but it has a "head" that reads the cells. The head can move left, right, or stay where it is. We have a diagram of it like this:



Your own computer is also like a fixed program with memory, except that the kind of memory is different. However, the typical input when you start your computer is huge. That input includes your operating system (such as Windows), other programs, and everything on your hard disk. You can give it additional input on CDs or online.

2 A sample program

Here's the code for a Turing machine program that takes an input in binary and adds one to it, then accepts and leaves the result on the tape:

STATE 0

IF READ 0, WRITE 0, MOVE right, GO TO STATE 0
IF READ 1, WRITE 1, MOVE right, GO TO STATE 0
IF READ □, WRITE □, MOVE left, GO TO STATE 1

STATE 1

IF READ 0, WRITE 1, KEEP POSITION, GO TO STATE 4
IF READ 1, WRITE 0, MOVE left, GO TO STATE 1
IF READ ▷, WRITE ▷, MOVE right, GO TO STATE 2

STATE 2

IF READ 0, WRITE 1, MOVE right, GO TO STATE 3
IF READ 1, WRITE 1, MOVE right, GO TO STATE 2

IF READ \square , WRITE 1, KEEP POSITION, GO TO STATE 4

STATE 3

IF READ 0, WRITE 0, MOVE right, GO TO STATE 3

IF READ 1, WRITE 0, MOVE right, GO TO STATE 2

IF READ \square , WRITE 0, KEEP POSITION, GO TO STATE 4

STATE 4

HALT and ACCEPT

Let's think about the stages of adding one to a binary number. If you add one to 11011, the result is 11100. Intuitively, what you do is start at the right, change each 1 to a 0 until you hit a 0, and change that 0 to a 1. Your only hope to understand how this program works is to run through it; after you've run through it once or twice on your own, you can read this description to make sure you're understanding correctly:

- STATE 0 moves the head of the Turing machine to the right until it gets to the end of the input.
- STATE 1 moves left, replacing each 1 with a 0, until it hits a 0 itself; if it hits that 0, it replaces it with a 1 and we're done. Thus it goes to STATE 4.

But what if we never hit a 0? What if our input is 11111? Then the proper result should be 100000. So STATE 1 also checks to see if we ever get to the left end of the tape. If so, we need to start shifting everything to the right to make space for the starting 1! So we go to STATE 2.

- STATE 2 and STATE 3 work together. Each one works to shift the input one step to the right. Suppose the head is currently reading 0. Then it needs to write whatever character it was at one move before, and remember to write 0 in the next spot over.

STATE 2 is where we go if the last character we read was a 1; that means we have to write 1 in the spot we're at now.

STATE 3 is where we go if the last character we read was a 0; then we have to write 0 in the spot we're at now.

If STATE 2 reads a 0, then it goes to STATE 3 (after moving right) — that's the Turing machine's way of remembering "I need to write a 0 next." If STATE 2 reads a 1, then it stays at STATE 2 (after moving right) — that's the Turing machine's way of remembering "I need to write a 1 next."

That's how STATE 2 and STATE 3 work together to move the input one spot to the right. By starting at STATE 2, we tell the machine to write a 1 first thing; that's how we get the starting 1 in the final output.

- STATE 4 is the halting state. Since every Turing machine must accept or reject, we accept by default.

3 What a Turing machine is

A **Turing machine** is just a program like the one above — that’s everything you need to state what a Turing machine is. The infinite tape is *assumed* — it’s not part of what a Turing machine “is.”

We say that a Turing machine **accepts** its input if starting with its input on an infinite tape and running the program, you end in an **ACCEPT** state. We say that a Turing machine **rejects** its input if you end in a **REJECT** state. Finally, we say that a Turing machine **loops** on a particular input if it never enters a halt state while computing on that input.¹

We want to turn these concepts into good formal definitions. This seems bothersome at first, but it will prove to be really useful.

3.1 Formal definition of a Turing machine

We’ll encode a “program” like the one above in terms of sets (so it’s a purely mathematical object). We’ll have a **set of states** that records what states we can go to. The lines of code that say **IF READ 0, WRITE 0, MOVE RIGHT, GO TO STATE 3** will all be recorded in the **transition function**.

Definition 1. A **Turing machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where Q , Σ , and Γ are finite sets and:

1. Q is the **set of states**.
2. Σ is the **input alphabet** not containing the special **blank symbol** \sqcup or the special **tape marker** \triangleright .
3. Γ is the **tape alphabet** with $\sqcup \in \Gamma, \triangleright \in \Gamma$ and $\Sigma \subset \Gamma$.
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$ is the **transition function**. For each $q \in Q$, we also specify that $\delta(q, \triangleright) = (q', \triangleright, R)$ for some $q' \in Q$.
5. $q_0 \in Q$ is the **start state**.
6. $q_{\text{accept}} \in Q$ is the **accept state**.
7. $q_{\text{reject}} \in Q$ is the **reject state**.

In the program above, we would have $Q = \{0, 1, 2, 3, 4, 5\}$ (we need 5 to be a reject state, since technically the definition of a Turing machine requires it). $\Sigma = \{0, 1\}$; the Turing machine only takes input in binary. $\Gamma = \{0, 1, \triangleright, \sqcup\}$; we add \triangleright and \sqcup because the Turing machine can read and write these symbols, but they’re not allowed in the input.

$q_0 = 0$ because we start in **STATE 0**. $q_{\text{accept}} = 4$ because **STATE 4** is the state that halts and accepts. $q_{\text{reject}} = 5$ because we need a reject state; you can’t ever end up in **STATE 5**, but it fulfills the definition.

¹The term “loops” is misleading — this isn’t necessarily even a loop! A machine might go on forever writing more and more 1s on the tape; that’s not an infinite loop because the state is changing, but it still runs forever, so we still say the machine loops!

δ is complicated. It's a function that takes in a state and a character on the tape, and outputs a new state (the state we go to), a new character (what we write on the tape), and a direction to move the tape head (left, right, or stay put).

Recall that we had:

STATE 0

```
IF READ 0, WRITE 0, MOVE right, GO TO STATE 0
IF READ 1, WRITE 1, MOVE right, GO TO STATE 0
IF READ  $\sqcup$ , WRITE  $\sqcup$ , MOVE left, GO TO STATE 1
```

This tells us that $\delta(0, 0) = (0, 0, R)$ — if we're in state 0 and we read a 0, then go to state 0 after writing a 0 and moving right. Similarly, $\delta(0, 1) = (0, 1, R)$ and $\delta(0, \sqcup) = (1, \sqcup, L)$.

From

STATE 1

```
IF READ 0, WRITE 1, KEEP POSITION, GO TO STATE 4
IF READ 1, WRITE 0, MOVE left, GO TO STATE 1
IF READ  $\triangleright$ , WRITE  $\triangleright$ , MOVE right, GO TO STATE 2
```

We get

$$\begin{aligned}\delta(1, 0) &= (4, 1, S) \\ \delta(1, 1) &= (1, 0, L) \\ \delta(1, \triangleright) &= (2, \triangleright, R)\end{aligned}$$

We could go on and do this for all the other states, too. The upshot is: **the formal definition encodes all the information our “programs” do, but in a formal mathematical sense.**

3.2 The definition of computation

There's one more thing we don't really have defined. We know intuitively how a Turing machine computes on an infinite tape, but we haven't defined this formally and mathematically. That's the next step.

Our definition of computation will monitor everything to make sure that a Turing machine goes step-by-step through given input.

3.3 Step 1: recording what the tape looks like at one point in time

What is a “step?” A step is something that goes from one **configuration** to another **configuration**. A configuration must encode all of the information about the current state of the Turing machine: it must tell you where the head of the machine is and what's on the tape. If you know that, and you had the description of the Turing machine itself, you could keep going with the computation.

Definition 2. A **configuration** of a Turing machine is a string uqv where $u, v \in \Gamma^*$ (i.e. u and v are strings composed of characters from Γ) and $q \in Q$ (i.e. q is a state of the Turing machine).

For example, $\triangleright 01101q_8 0101$ is a configuration of a Turing machine. This says that the tape currently reads $\triangleright 011010101$ and that the Turing machine is in state q_8 . It also says that the head is currently sitting on the 7th spot on the tape, reading a 0. See how all the information about the current status of a Turing machine is compactly recorded?

3.4 Step 2: how to go forward “one step”

If we know how to go forward one step, we can keep going one step at a time until we get to the end and we know if we accept or reject.

Definition 3. A configuration **yields** another if it follows from δ . More precisely, for $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, and $q_i, q_j \in Q$:

1. $uaq_i bv$ yields $uq_j acv$ if $\delta(q_i, b) = (q_j, c, L)$.
2. $uaq_i bv$ yields $uacq_j v$ if $\delta(q_i, b) = (q_j, c, R)$.
3. $uaq_i bv$ yields $uaq_j cv$ if $\delta(q_i, b) = (q_j, c, S)$.

3.5 Step 3: How we get the final answer

A Turing machine accepts an input if it ends in an **accepting configuration**.

Definition 4. An **accepting configuration** is a configuration with $q = q_{\text{accept}}$.

To figure out if it ever gets to such a state, we look at accepting computations:

Definition 5. A Turing machine M **accepts** input w if there exist C_1, C_2, \dots, C_k configurations of M with:

1. $C_1 = \triangleright q_0 w$.
2. Each C_i yields C_{i+1} .
3. C_k is an accepting configuration.

3.6 Step 4: Rephrasing everything we know in terms of these new definitions

The amazing thing about Turing machines is that, although they’re so simple, they have all the power of the computers we work with regularly. In fact, a programming language is called **Turing-complete** if it can do everything a Turing machine can do.

Even though Turing machines are so powerful, they’re also very simple. So it’s easy to work with them in mathematical proofs and conclude things about what can and cannot be

done by computers. Now we can take everything we've defined before and see how it relates to Turing machines. This will be the first time we'll actually give mathematically rigorous definitions.

Definition 6. A Turing machine M **recognizes** a language L if $w \in L \Leftrightarrow M$ accepts w .

Definition 7. A Turing machine **halts** on input w if there do not exist C_1, C_2, \dots such that:

1. $C_1 = \triangleright q_0 w$.
2. Each C_i yields C_{i+1} .
3. No C_i is an accepting or rejecting configuration.

Definition 8. A Turing machine M **decides** a language L if M recognizes L **and** M halts on each input.

3.7 Step 5: The languages we know

Here are languages we were familiar with, now redone:

$$A_{\text{TM}} = \{ \langle P, w \rangle \mid P \text{ is a Turing machine and } P \text{ accepts } w \}$$

$$\text{HALT}_{\text{TM}} = \{ \langle P, w \rangle \mid P \text{ is a Turing machine and } P \text{ halts on } w \}$$

4 Wrapping it up

With Turing machines, our definition of computation is now rigorous, and their simplicity will allow us to prove lots of stuff. Turing machines are entirely mathematical objects.

In practice, the knowledge that Turing machines are essentially just as powerful as computers means that when we design Turing machine programs we won't sit down and write out all the states. Pseudocode is still enough, so long as it's precise enough to convince us that a Turing machine could easily be created to do the same thing. This principle will mean that once we've done a few more things with Turing machines, we won't need them very often to continue to build up our theoretical framework.