Announcements

1. All midterms due by early Sunday 5PM.

Turing Machines

1. Last class, I introduced Turing machines with an informal example and discussed how we will represent TMs as drawings.

2. I used a Turing machine from Sipser (Example 3.9) that is designed to recognize strings of the form \( w\#w \) to introduce informal notation used to describe Turing machines and to explain how they work.

3. I showed a hand-drawn Turing machine that recognizes a language quite similar to the language Sipser’s machine recognizes.

   • Instead of \( w\#w \), this machine recognizes \( ww \).

   • Instead of using the binary alphabet \( \{0,1\} \) as Sipser did, this machine uses \( \{a,b\} \). Of course, the shapes of the symbols don’t really matter.

4. I would like to illustrate another way we could build a machine that recognizes \( ww \), this time over the alphabet \( \{0,1\} \). The idea is to re-use Sipser’s machine by doing a little pre-processing.

5. Consider the machine:

    convert \( ww \) to \( w\#w \)

   6. This machine tries to interpret its input as a string that might be of the form \( ww \) and convert it to a string of the form \( w\#w \) by inserting a \# at the midpoint of its input.

Click here to view the slides for this class
• It first marks and remembers the left-most input character.
  – Since our tape alphabet is unlimited, we can assume the existence of primed, double-primed, hatted, etc. copies of every symbol in our input alphabet.
• It then scans to the end of the tape remembering whether it has seen an even or odd number of symbols, and what the last symbol in an even location was.
• When it hits the end of the tape, it rejects if it has seen an odd number of symbols. If it has seen an even number, it writes the last symbol seen over the first blank and then back up to the original copy of the symbol.
• Next it puts a # in place of what had been the last symbol.
• Now, it enters its main loop. It repeats this as long as there are more unmarked symbols to the left of the position where it has most recently placed the #.
  – It scans left to find the rightmost marked symbol.
  – It then remembers and marks the unmarked symbol just to the right of the rightmost marked symbol and begins a scan to the right.
  – It scan until it hits the # remembering the last symbol it saw
  – When it finds the #, it interchanges the # with the symbol that appeared just before the #.
• When this loop terminates, the # has been positioned correctly, but all the symbols to its left are marked. The final loop makes a pass to the left unmarking all these symbols. It stops by taking advantage that in Sipser’s version of TMs, if you attempt to move off the left end of the tape, you just remain on the left-most symbol.

7. Such a machine is called a transducer. Its purpose is not really to accept or reject strings. Its purpose is to transform strings. If the arrow leading from the right side is connect to the start state of Sipser’s machine, the combined machine will only accepts strings of the form $ww$.

8. This illustrates a useful way to think about designing and describing complex Turing machines. Just as we break real programs up into functions or methods, we can break Turing machines up into sub-modules to perform certain transformations or checks on sub-parts of the input.

9. Here is another interesting module:

Given any input string of as and bs, this collection of states acts like a Xerox machine. It makes a second copy of the input following the first separated from the first by a space.

• The letter states are used to copy an a from the original input to the first blank after the incomplete copy.
• The numbered states are used to copy a b.
• Each set of states first marks the symbol being copied by replacing it with an x and then scans until it finds a blank.
• Next they skip over any partial copy until another blank is found.
• The second blank is replace by the copied character.
• Finally, the machine scans back to find the marked symbol that was just copied, unmarks it and starts with the next symbol.

10. With a bit of imagination, you can think of this machine as a primitive assignment statement. After all, “x = y;” makes a copy of y’s value. OK, maybe it takes a lot of imagination.

11. As another example of a fragment of a Turing machine that might be a useful component in a bigger structure, consider the machine:

\[
\begin{align*}
S' & \quad 0,1 \rightarrow R \\
# & \rightarrow L \\
B & \quad 0 \rightarrow 1, L \\
1 & \rightarrow 0, L
\end{align*}
\]

• This machine is simple enough that you should be able to figure out its function on your own?
• Did you? If not, it interprets its input as a string of the form \(n\#\ldots\) where \(n\) is the representation of some positive number represented in binary notation, and it transforms its input to be \(n-1\#\ldots\).

12. With this little sub-module, let’s think about how to construct a Turing machine to recognize the language:

\(\{i\#x\#w_1\#w_2\#\ldots\#w_k \mid i, x, w_i \in \{0, 1\}^*, 0 < i \leq k, \text{ and } x = w_n\}\)

which corresponds roughly to the computation associated with a statement like “if \((x = w[i])\) ...”
• The trick here is that instead of worrying about all the details, we will view the machine we have looked at in detail as enough to give us confidence that we can create sub-modules that
  – subtract 1 from a sub-string of the input interpreted as a binary number,
  – determine if two clearly delimited substrings are identical (as we did for \(w\#w\)).
• The machine we would design would first turn the first two \#’s into some other marker.
• Next, it would repeatedly scan out to the second special marker, replace it with a plain old \#, replace the next \# to the right with the special marker
• Then it would go back and replace \(n\) by \(n-1\) and repeat the marker moving trick over and over until \(n\) became 0.
• After this was complete, the two substrings that should be compared would be marked with special markers.
• It should be clear we could adapt Sipser’s machine to check that these two substrings were identical.

13. This strange language is supposed to remind you of something familiar: arrays in most programming languages or the array which is a processor’s memory. The point is that even with little experience with Turing machines, we can begin to see that it is possible to program these simple machines in ways that resemble features of typical programs (like keeping an array of values and indexing the array).

**Getting Formal**

1. We can formalize our understanding Turing Machines, with a few exciting definitions:

**Definition:** A Turing machine is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where

\(Q\) is a finite set of states,
\(\Sigma\) is a finite input alphabet (not containing the blank symbol),
\(\Gamma\) is a finite tape alphabet which is a superset of \(\Sigma\) including the blank symbol,
\[ \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{Left, Right}\} \]

is the transition function,

\( q_0 \) is the start state,

\( q_{\text{accept}} \) is the accept state, and

\( q_{\text{reject}} \neq q_{\text{accept}} \) is the reject state.

2. Next, to capture the way we expect a Turing Machine to compute, we need a precise way of describing a snapshot of a point in some computation.

**Definition:** A configuration of a Turing machine is a triple \((u, q, v)\) where \(q \in Q\) is the current state, \(uv\) is the contents of the non-blank portion of the tape with \(u\) being the portion to the left of the current head position and \(v\) being the portion from the symbol currently under the head to the end of the non-blank tape.

3. With these definitions, we can now state exactly what it means for a Turing Machine to accept a string.

**Definition:** We say the configuration \((u, q, av)\) yields configuration \((u', q', v')\) for \(q, q' \in Q, a \in \Gamma,\) and \(u, v, u', v' \in \Gamma^*\) if for some \(b\) and \(c\) \(\in \Gamma:\)

- \( \delta(q, a) = (q', c, \text{Left}), u = u'b, \) and \( v' = bcv, \) or
- \( \delta(q, a) = (q', c, \text{Right}), u' = uc \) and \( v' = v, \) or
- \( \delta(q, a) = (q', c, \text{Left}), u = u' = \epsilon, \) and \( v' = cv, \) or
- \( \delta(q, a) = (q', c, \text{Right}), u' = uc, v = \epsilon, \) and \( v' = _. \)