Announcements
1. All midterms due by early Sunday morning.

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

2. To continue this discussion, consider how the machine shown below (and last class and discussed in Sipser) would behave if its input tape initially contained the string “01#01” followed by blank tape.

3. While I love state diagrams, Turing machines quickly get large and complicated, so it is often useful to give a less formal description of their operation. The machine above, for example can be described as follows.

(a) If the symbol under the read head is a #, scan to the right and if you only find x’s until you reach blank tape accept. Otherwise,

(b) Remember the character you start at and replace it with an x.

(c) Scan to the right until you find a #.

(d) Continue scanning to the right skipping any x’s.

(e) If the first symbol after the x’s does not match what you saw in state b, reject.

(f) If the symbol after the x’s matches, replace it with an x.

(g) Scan to the left passing all symbols until you get back to the #.

(h) Keep scanning to the left to find the first x.

(i) Move to the right and return to state a.

4. I found a hand-drawn Turing machine using Google images 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.

5. 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.

6. Consider the machine:

7. 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.

• 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 write 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.

8. 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 $ww$.

9. 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.

10. Here is another interesting module:

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

\[ S' \quad 0,1 \rightarrow R \\
\quad \textbf{B} \quad \text{#} \rightarrow L \\
\quad 0 \rightarrow 1, L \\
\quad 1 \rightarrow 0, L \]

- This machine is simple enough that you should be able to figure out its function on your own?
- Did you? If not, it interprets is 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 \( n - 1 \# \ldots \).

13. 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.

14. 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).