Announcements
1. Last chance to pick up midterms today.

Turing Machines

1. Last time we explored/sketched out several Turing machines informally:
   - Sipser’s w#w machine.
   - One to recognize ww by taking advantage of the machine Sipser provided to recognize w#w using a sub-module that inserted a # in the middle.
   - A fragment of a machine to subtract 1 from a binary number.
   - Another to recognize \( \{ i \# x \# w_1 \# w_2 \# \ldots \# w_k \mid i, x, w_i \in \{0, 1\}^*, i \leq k, \text{ and } x = w_i \} \)

2. We also began to discuss how to formalize our understanding of Turing Machines, with a few exciting definitions:

   **Definition:** A Turing machine is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})\), 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,
   - \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{Left, Right\}\) is the transition function,
   - \(q_0\) is the start state,
   - \(q_{accept}\) is the accept state, and
   - \(q_{reject} \neq q_{accept}\) is the reject state.

3. Our first goal today is to finish off the formal definitions needed to capture what it means for a Turing machine to describe a language.

   **Definition:** We say the configuration \((u, q, \epsilon)\) yields configuration \((u', q', v)\) for \(q, q' \in Q, a \in \Gamma, \text{ and } u, v, u', v' \in \Gamma^*\) if for some \(b\) and \(c\) in \(\Gamma\):
   - \(\delta(q, a) = (q', c, Left), u = u'b, \text{ and } v' = bcv, \text{ or }\)
   - \(\delta(q, a) = (q', c, Right), u' = uc \text{ and } v' = v, \text{ or }\)
   - \(\delta(q, a) = (q', c, Left), u = \epsilon, \text{ and } v' = cv, \text{ or }\)
   - \(\delta(q, a) = (q', c, Right), u' = uc', v = \epsilon, \text{ and } v' = \epsilon\).

   **Definition:** A TM accepts a string \(w \in \Sigma^*\) if there is a sequence of configurations that begins with \((\epsilon, q_0, w)\) and ends in \((w', q_{accept}, w'')\) for some \(w', w'' \in \Gamma^*\) where each configuration yields the following configuration in the series.

4. If Turing machines were like DFAs or PDAs, we would be done now.

   **Definition:** A language \(L\) is Turing-recognizable if some Turing machine accepts \(w\) if and only if \(w \in L\). We call these languages recursively-enumerable.

   **Definition:** A language \(L\) is Turing-decidable if some Turing machine that halts on all inputs accepts \(w\) if and only if \(w \in L\). We call these languages recursive.
Variations on Automata

1. Last class, I tried to explain that part of our agenda at this point is to explore the case that the Turing Machine is a model that captures everything a computer can do. We cannot prove this, but we can reassure ourselves that it seems reasonable to make this assumption in two ways.

- First, we will explore whether adding features to the Turing Machine model adds computational power.
  - If adding features does not increase the computational power of the model, maybe the model is as powerful as it could be!
  - For example, we have already done a bit of this with DFAs. We saw (surprisingly?) that adding nondeterminism to our model for DFAs did not extend its computational power. On the other hand, if we changed the DFA model so that a DFA could move back and forth and write on its tape as long as it did not go beyond the space originally filled with its input, this would increase the power of the model.

- In addition, we can argue that the Turing Machine model is computationally equivalent to several other models that are radically different in design.
  - I have already hinted broadly that a TM tape could be used as a big array suggesting it might be possible to simulate computations in traditional programming systems (Java?) that use arrays.

2. As a starting point, rather than adding a feature to the TM model, let’s consider the impact of adding a feature to one of our less powerful models, the PDA. What happens to the computational power of a PDA if we give it two stacks?

2-PDAs

1. Recall the formal definition of a pushdown automaton:

   **Definition:** A pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where:

   - \(Q\) is a finite set of states,
   - \(\Sigma\) is a finite input alphabet,
   - \(\Gamma\) is a finite stack alphabet,
   - \(\delta : Q \times \Sigma \times \epsilon \rightarrow \mathcal{P}(Q \times \Gamma)\) is the transition function, and
   - \(F \subset Q\) is the set of final or accepting states.

2. Now, we can define a 2-tape PDA:

   **Definition:** A pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where:

   - \(Q\) is a finite set of states,
   - \(\Sigma\) is a finite input alphabet,
   - \(\Gamma\) is a finite stack alphabet,
   - \(\delta : Q \times \Sigma \times \Gamma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \Gamma)\) is the transition function, and
   - \(F \subset Q\) is the set of final or accepting states.

3. The diagram below is an example of the specification of a 2-tape PDA.

   - If a transition arrow is labeled “a, b, c / d, e” it means that if the machine is in the state where the arrow originates, it can transition to the target of the arrow if a is the next input character, b and c are the characters at the tops of its two states. If the transition is used, then the symbols b and c should be popped from the stacks and d and e should be pushed. Any of a, b, c, d, and e can be empty.
• While in state pre #, the machine shown scans until it finds a # while pushing all of the as and bs it encounters onto its first stack so that at the end the first symbol from the input is at the bottom of the first stack and the last is at the top of the first stack.

• When it hits the #, it pops each symbol from its first stack and pushes it on the second stack until the first stack is empty. When this is complete, the second stack contains a copy of the first half of the input with the first symbol at the top of the stack and the last symbol at the bottom.

• Next, it scans to the right popping symbols off the second stack as long as each symbol on the stack matches the next input symbol.

• If the second half of the input matches the first, it will eventually empty the second stack and then transition to its only accepting state. Thus, the machine is designed to accept $w\#w$.

4. Remember, $w\#w$ is not a context-free language. The machine we just described therefore shows that 2-tape PDAs are clearly more powerful than single stack PDAs.

5. To appreciate just how much more powerful a 2-tape PDA is, recall the notion of a Turing machine configuration:

**Definition:** A configuration of a Turing machine is a triple $(u, q, v)$ where $q \in Q$ is the current state, $wv$ 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.

• Given a TM configuration, we could store all the symbols that are before the tape head ($u$) in one stack and all of the symbols after the tape head (including the symbol under the tape head) in the second stack.

• We could then define the remaining states and transitions of the 2-tape PDA in a way that mimicked any TM. In particular, if $\delta_T$ is the TM’s transition function and $\delta_P$ is the transition function for our PDA then

  - If $\delta_T(q,a) = (q’, b, Right)$ then $\delta_P(q’,\epsilon,\epsilon,a) = \{(q’, b, \epsilon)\}$.
  - If $\delta_T(q,a) = (q’, b, Left)$ then for all $c$,
    * $\delta_P(q’,\epsilon,\epsilon,a) = \{(q’-push,\epsilon,b)\}$, and
    * $\delta_P(q’-push,\epsilon,c,\epsilon) = \{(q’,\epsilon,c)\}$(the intermediate state $q’$-push-c is only required because our definition of PDA’s limits us to pushing one symbol on the stack at a time).

Thus, a 2-tape PDA is at least as powerful as a Turing Machine!

**n-tape Turing Machines**

1. What happens to the computational power of a TM if we give it more than one tape?

2. That is, we stick with one finite control that will be in a single state at any point, but we give the machine n-tapes and one read head that can be independently positioned for each tape as suggested by the figure below:

3. Formally, we would say
**Definition:** A n-tape 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,
- **\(\delta: Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{\text{Left, Right}\}^n\)** 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.

That is, each transition would be determined by the set of \(n\) symbols currently under the read heads for each tape and would determine \(n\) independent symbols to write and \(n\) independent moves left or right on the tape.

To make this formalism complete, we would also have to extend the notions of a TM configuration and a valid, accepting sequence of configurations.

4. Such a machine might or might not be able to compute things that a Turing machine cannot compute, but it is much easier to program. To see this consider the following example.

- At the risk of making you think it is the only language any TM can decide, I will again use \(w\#w\).
- This language does not justify a machine with many tapes, but it is definitely easier to recognize \(w\#w\) on a 2-tape TM than on a single-tape TM.
- On a single-tape TM, recognizing \(w\#w\) requires making \(|w|\) passes back and forth from one copy of \(w\) to the other \(w\) marking matching symbols to verify that each symbol has a match.
- On a 2-tape TM, we can complete the entire process in one and a half passes:

```plaintext
- The machine would start with the input on its first tape and nothing on the other.
```

```plaintext
- It would first scan the input from left to right copying symbols from the input tape to the machine’s second tape until reaching a #. This is the role of state \(C = \text{COPY}\).
```
Next, it moves the head on the second tape back to the left end of the tape (We would like to keep the head on the first tape right where it is, but since my version of $n$-tape TMs require each head to move left or right at each step, we have to make it wiggle back and forth). This is the role of state $B = \text{BACKUP}$.

Finally, it scans right on both tapes at the same time making sure that the contents of the second tape matches the contents of the second half of the input/first tape. This is the role of state $M = \text{MATCH}$.

- For example, the other day we considered

$$\{i\#x\#w_1\#w_2\#\ldots\#w_k \mid i, x, w_i \in \{0, 1\}^*, i \leq k, \; \& \; x = w_i\}$$

- A 3-tape TM could recognize this language easily by first copying $i$ from the input tape to a second tape and next copying $w$ to its third tape. Then, as it moved to the right on its input tape it could decrement the value of $n$ on the second tape each time it hit a marker. When the counter became 0, it could match the $w$ on the third tape with $w_i$ on the input tape. It would never have to back up on its input tape!