Turing Machines
Part Two
Recap from Last Time
This part of the Turing machine is called the finite-state control. It issues commands that drive the machine.
Our First Turing Machine

This is the TM's infinite tape. Each tape cell holds a single tape symbol. Initially, all the tape symbols are blank.
Our First Turing Machine

The machine is started up with the input string written somewhere on the tape. The tape head initially points to the first symbol in the input.
Our First Turing Machine

Like DFAs and NFAs, the TM begins in its start state and starts reading transitions.
Our First Turing Machine

At each point in time, the TM only cares about the symbol directly under its tape head.
Our First Turing Machine

Each transition has the form

\[ \text{read} \rightarrow \text{write, DIR} \]

and means “if the current symbol is \text{read}, replace it with \text{write} and move in direction \text{DIR} (either \text{R} or \text{L}). The \[ \square \] symbol represents a blank cell.
Our First Turing Machine

This special state is an **accepting state**. When a TM enters an accepting state, it *immediately* stops running and accepts the input that was provided.
Our First Turing Machine

This special state is a **rejecting state**. When a TM enters a rejecting state, it *immediately* stops running and rejects the input that was provided.
Our First Turing Machine

If the TM is started with the empty string as input, the entire tape is blank and the tape head points to some arbitrary position.
New Stuff!
Main Question for Today:

Just how powerful are Turing machines?
Another TM Design

- Last time, we designed a TM for this language over $\Sigma = \{0, 1\}$:
  
  $$L = \{ w \in \Sigma^* \mid w \text{ has the same number of } 0\text{s and } 1\text{s } \}$$

- Let's do a quick review of how it worked.
The Solution

... × 0 0 × 1 1 1 0 ...

...
A Different Idea
A Different Strategy

Last time, we built a machine that checks whether a string has the form $0^n1^n$. That machine almost solves this problem, but requires that the characters have to be in order.

What if we sorted the input?
A Different Strategy

Observation 1: A string of 0s and 1s is sorted if it matches the regex \(0^*1^*\).
A Different Strategy

Observation 2: A string of 0s and 1s is not sorted if it contains 10 as a substring.
A Different Strategy

Idea: Repeatedly find a copy of 10 and replace it with 01.
Let's Build It!
This is just a placeholder. Imagine snapping in the entire TM for $0^n1^n$ into this diagram, putting the start state in the dashed area.
This TM will sort any sequence of 0s and 1s, but it might take a while.

**Fun problem:** design a TM that sorts a string of 0s and 1s, but does so while taking way fewer steps than this machine.
TM Subroutines

- A **TM subroutine** is a Turing machine that, instead of accepting or rejecting an input, does some sort of processing job.
- TM subroutines let us compose larger TMs out of smaller TMs, just as you'd write a larger program using lots of smaller helper functions.
- Here, we saw a TM subroutine that sorts a sequence of 0s and 1s into ascending order.
TM Subroutines

• Typically, when a subroutine is done running, you have it enter a state marked “done” with a dashed line around it.

• When we're composing multiple subroutines together – which we'll do in a bit – the idea is that we'll snap in some real state for the “done” state.
What other subroutines can we make?
Let's design a TM that, given a tape that looks like this:

```
... 1 3 7 4 2 ...
```

ends up having the tape look like this:

```
... 1 7 9 0 0 ...
```

In other words, we want to build a TM that can add two numbers.
There are many ways we could in principle design this TM.

We're going to take the following approach:

- First, we'll build a TM that increments a number.
- Next, we'll build a TM that decrements a number.
- Then, we'll combine them together, repeatedly decrementing the second number and adding one to the first number.
Incrementing Numbers

• Let's begin by building a TM that increments a number.

• We'll assume that
  – the tape head points at the start of a number,
  – there are at least two blanks to the left of the number, and
  – that there's at least one blank at the start of the number.

• The tape head will end at the start of the number after incrementing it.
Incrementing Numbers

```plaintext
go to the end of the number;
while (the current digit is 9) {
  set the current digit to 0;
  back up one digit;
}
increment the current digit;
go to the start of the number;
```
0 → 0, R
1 → 1, R
...
9 → 9, R

start

To End

□ → □, L

Wrap Nines

9 → 0, L

0 → 1, L
1 → 2, L
2 → 3, L
...
8 → 9, L
□ → 1, L

done!

□ → □, R

Back Home

0 → 0, L
1 → 1, L
...
9 → 9, L
Decrementing Numbers

- Now, let's build a TM that decrements a number.
- We'll assume that
  - the tape head points at the start of a number,
  - there is at least one blank on each side of the number.
- The tape head will end at the start of the number after decrementing it.
- If the number is 0, then the subroutine should somehow signal this rather than making the number negative.
Decrementing Numbers

go to the end of the number;
if (every digit was 0) {
  signal that we're done;
}
while (the current digit is 0) {
  set the current digit to 9;
  back up one digit;
}
decrement the current digit;
go to the start of the number;
0 → 0, R
1 → 1, R
2 → 2, R
3 → 2, L
4 → 1, L
5 → 0, L
6 → 9, L
7 → 8, L
8 → 7, L
9 → 9, R

0 → 0, R
1 → 1, R
2 → 2, R
3 → 2, L
4 → 1, L
5 → 0, L
6 → 9, L
7 → 8, L
8 → 7, L
9 → 9, R

0 → 0, R
1 → 1, R
2 → 2, R
3 → 2, L
4 → 1, L
5 → 0, L
6 → 9, L
7 → 8, L
8 → 7, L
9 → 9, R

\[ n = 0 \]

done!

Back Home

Wrap Zeros

0 → 0, L
1 → 1, L
2 → 1, L
3 → 2, L
4 → 1, L
5 → 0, L
6 → 9, L
7 → 8, L
8 → 7, L
9 → 9, L

Non-zero?

\[ \square \rightarrow \square, \ L \]

To End

\[ \square \rightarrow \square, \ L \]
TM Subroutines

- Sometimes, a subroutine needs to report back some information about what happened.
- Just as a function can return multiple different values, we'll allow subroutines to have different “done” states.
- Each state can then be wired to a different state, so a TM using the subroutine can control what happens next.
Putting it All Together

- Our goal is to build a TM that, given two numbers, adds those numbers together.

- Before:
  
  ... [1 3 7 4 2] ...

- After:
  
  ... [1 7 9 0 0] ...
Using Our Subroutines

- We'll build our new machine using our existing increment and decrement subroutines:
Using Subroutines

- Once you've built a subroutine, you can wire it into another TM with something that, schematically, looks like this:

- Intuitively, this corresponds to transitioning to the start state of the subroutine, then replacing the “done” state of the subroutine with the state at the end of the transition.
Time-Out for Announcements!
Problem Set Logistics

- Problem Set 7 was due at the start of class today.
  - Using late days, can extend that deadline up to Monday, November 21st.
- Problem Set 8 goes out today. It's due on the Friday we get back (December 2).
  - Explore context-free grammars and languages!
  - Play around with Turing machines!
- Some of the problems on that problem set require content from Monday's lecture. Those parts are clearly marked (and, IMHO, the easier parts of the problem set).
Your Questions
“Tell us about the T shirt that you're wearing today-what's the story behind the organization? When did you hear about it and have you showed support in other ways?”

It's the South African National Rugby Team shirt. It's a longish sort of story.
“Tell us something interesting that has nothing to do with CS103.”

With Thanksgiving coming up, let’s talk about fruits and vegetables that are native to the Americas!
Back to CS103!
Main Question for Today:

Just how powerful are Turing machines?
How Powerful are TMs?

• Regular languages, intuitively, are as powerful as computers with finite memory.

• TMs by themselves seem like they can do a fair number of tasks, but it's unclear specifically what they can do.

• Let's explore their expressive power.
Real and “Ideal” Computers

- A real computer has memory limitations: you have a finite amount of RAM, a finite amount of disk space, etc.
- However, as computers get more and more powerful, the amount of memory available keeps increasing.
- An *idealized computer* is like a regular computer, but with unlimited RAM and disk space. It functions just like a regular computer, but never runs out of memory.
Claim 1: Idealized computers can simulate Turing machines.

“Anything that can be done with a TM can also be done with an unbounded-memory computer.”
The TM's finite-state control can be encoded as a table, making it easy for a computer to look up transitions information.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>□</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>q₀</strong></td>
<td><strong>q₁</strong></td>
<td>□</td>
<td><strong>R</strong></td>
</tr>
<tr>
<td><strong>q₁</strong></td>
<td>□</td>
<td><strong>R</strong></td>
<td><strong>q₁</strong></td>
</tr>
<tr>
<td><strong>q₂</strong></td>
<td><strong>qᵣ</strong></td>
<td>□</td>
<td><strong>R</strong></td>
</tr>
<tr>
<td><strong>q₃</strong></td>
<td>□</td>
<td><strong>L</strong></td>
<td><strong>q₃</strong></td>
</tr>
</tbody>
</table>
Simulating a TM

- To simulate a TM, the computer would need to be able to keep track of:
  - the finite-state control,
  - the current state,
  - the position of the tape head, and
  - the tape contents.
- The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
- We only need to store the “interesting” part of the tape (the parts that have been read from or written to so far.)
Claim 2: Turing machines can simulate idealized computers.

“Anything that can be done with an unbounded-memory computer can be done with a TM.”
What We've Seen

- TMs can
  - implement loops (basically, every TM we've seen).
  - make function calls (subroutines).
  - keep track of natural numbers (written in unary or in decimal on the tape).
  - perform elementary arithmetic (equality testing, multiplication, addition, increment, decrement, etc.).
  - perform if/else tests (different transitions based on different cases).
What Else Can TMs Do?

- Maintain variables.
  - Have a dedicated part of the tape where the variables are stored.
  - We've seen this before: take a look at our machine for composite numbers, or for increment/decrement.

- Maintain arrays and linked structures.
  - Divide the tape into different regions corresponding to memory locations.
  - Represent arrays and linked structures by keeping track of the ID of one of those regions.
A CS107 Perspective

- Internally, computers execute by using basic operations like
  - simple arithmetic,
  - memory reads and writes,
  - branches and jumps,
  - register operations,
  - etc.
- Each of these are simple enough that they could be simulated by a Turing machine.
A Leap of Faith

• It may require a leap of faith, but anything you can do a computer (excluding randomness and user input) can be performed by a Turing machine.

• The resulting TM might be colossal, or really slow, or both, but it would still faithfully simulate the computer.

• We're going to take this as an article of faith in CS103. If you curious for more details, come talk to me after class.
Just how powerful are Turing machines?
Effective Computation

- An effective method of computation is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.

- This is not a formal definition. Rather, it's a set of properties we expect out of a computational system.
The *Church-Turing Thesis* claims that every effective method of computation is either equivalent to or weaker than a Turing machine.

“This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!”

- Ryan Williams
Regular Languages \subset CFLs \subset Problems Solvable by Any Feasible Computing Machine
Regular Languages

CFLs

Problems solvable by Turing Machines

All Languages
TMs ≈ Computers

• Because Turing machines have the same computational powers as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.

• Going forward, we're going to switch back and forth between TMs and computer programs based on whatever is most appropriate.

• In fact, our eventual proofs about the existence of impossible problems will involve a good amount of pseudocode. Stay tuned for details!