Turing Machines
Part Two
Outline for Today

- **Recap from Last Time**
  - Where are we, again?

- **TM Subroutines**
  - Building larger TMs from smaller ones.

- **The Church-Turing Thesis**
  - What can you do with a TM?

- **R and RE Languages**
  - Two fundamental classes of problems.
Recap from Last Time
Our First Turing Machine

This is the Turing machine’s finite state control. It issues commands that drive the operation of the machine.
Our First Turing Machine

This is the TM’s *infinite tape*. Each tape cell holds a *tape symbol*. Initially, all (infinitely many) tape symbols are blank.

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

Like DFAs and NFAs, TMs begin execution in their start state.
Our First Turing Machine

At each step, the TM only looks at the symbol immediately under the *tape head*. 
Our First Turing Machine

These two transitions originate at the current state. We’re going to choose one of them to follow.
Our First Turing Machine

Each transition has the form

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

and means “if symbol \textit{read} is under the tape head, replace it with \textit{write} and move the tape head in direction \textit{dir} (L or R). The $\square$ symbol denotes a blank cell.
Our First Turing Machine

Unlike a DFA or NFA, a TM doesn’t stop after reading all the input characters. We keep running until the machine explicitly says to stop.
Our First Turing Machine

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

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

If the TM is started on the empty string $\varepsilon$, the entire tape is blank and the tape head is positioned at some arbitrary location on the tape.
Input and Tape Alphabets

- A Turing machine has two alphabets:
  - An input alphabet $\Sigma$. All input strings are written in the input alphabet.
  - A tape alphabet $\Gamma$, where $\Sigma \subseteq \Gamma$. The tape alphabet contains all symbols that can be written onto the tape.

- The tape alphabet $\Gamma$ can contain any number of symbols, but always contains at least one blank symbol, denoted $\square$. You are guaranteed $\square \notin \Sigma$.

- At startup, the Turing machine begins with an infinite tape of $\square$ symbols with the input written at some location. The tape head is positioned at the start of the input.
Remember that all missing transitions implicitly reject.
start

Edge Case

Check $m \neq 0$

0 → 0, R

Unmark

x → 0, L

x → x, R

0 → 0, R

1 → 1, R

Accept!

→

◻ → ◻, R

0 → 0, L

□ → □, R

Next

0 → x, R

□ → □, L

1 → 1, L

× → x, R

1 → 1, R

To End

× → ×, R

× → ×, L

0 → 0, R

1 → 1, R

Cross off 1

→

◻ → ◻, R

0 → 0, L

□ → □, R

Back home

1 → □, L

Back home

□ → □, L

1 → 1, L

□ → □, R

1 → 1, L

□ → □, R

0 → 0, L

□ → □, R

0 → 0, R

□ → □, R

0 → 0, R

□ → □, R

0 → 0, R
New Stuff!
Main Question for Today:
Just how powerful are Turing machines?
TM Subroutines
Another TM Design

• On Wednesday, 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.
A Different Strategy

Could we sort the characters of this string?
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 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.

• The idea is that you’d then replace the dashed “done” state with the next piece of the construction.
Time-Out for Announcements!
Problem Sets

• Problem Set Seven was due at 2:30PM today.
  • Using late days, you can extend the deadline to this Sunday at 2:30PM.
• Problem Set Eight goes out today. It’s due Friday at 2:30PM.
  • Play around with CFGs and Turing machines!
  • We have online tools you’ll use to design, test, and submit your CFGs and TMs. Hope this helps!
Your Questions
“Which of your former students is doing the coolest things today after having left Stanford?”
Computer Science Career Panel

Hear from Stanford CS alumni about what they're doing now and how they've navigated their careers! Dinner will be served!

November 20 | Gates 219 | 5:45PM

Principal software engineer at Empower and former member of the team that fixed the healthcare.gov technical crisis

Product Designer at Quora and former cofounder and head of product at Polymer

Software Engineer on the Anti-Abuse Team at Pinterest

Software Engineer at VMware and former CS106S coordinator

RSVP using this link.
“Which do you recommend: taking higher-level CS classes that interest us early in undergrad, or waiting to finish the CS core before moving to higher-level ones?”

As long as you aren’t jumping prerequisites, I’d encourage you to explore and see what you’re excited about! If you’re curious to see what’s out there, by all means, go for it!

Another option: look for 500-level CS courses, which are designed as surveys of particular topics and usually are just graded on attendance. They’re great ways to check out what different fields are all about.
“You clearly put a lot into your classes. What excites you about teaching, and what advice do you have for someone who loves to teach but isn’t keen on academia?”

There are plenty of ways to get involved in teaching without diving into the academic deep end. Programs like TEALS partner software engineers with high school teachers to bring CS education to places that otherwise might not receive any. Companies like Khan Academy, Coursera, Udacity, etc. are based on the idea of getting knowledge out to other people in ways that will change their lives. And you can always run your own event at a community space.

For me, it’s a mix, but the joy of sharing beautiful ideas that can fundamentally change peoples’ lives is really hard to beat!
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 transition information.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>□</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>q₀</strong></td>
<td>q₁</td>
<td>□</td>
<td>R</td>
</tr>
<tr>
<td><strong>q₁</strong></td>
<td>q₁</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td><strong>q₂</strong></td>
<td>q_r</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td><strong>q₃</strong></td>
<td>q₃</td>
<td>0</td>
<td>L</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, 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 checking multiples.
- 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.
But...but...but...can a TM do...

“cat pictures?”

Sure! A picture is just a 2D array of colors, and a color can be represented as a series of numbers.
But...but...but...can a TM do...

“cat pictures?”
“cat videos?”

If you think about it, a video is just a series of pictures!
Any other ideas?
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

CFLs

Problems Solvable by Any Feasible Computing Machine

All Languages
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!
Strings, Languages, Encodings, and Problems
What problems can we solve with a computer?
What problems can we solve with a computer?

What does it mean to “solve” a problem?
The Hailstone Sequence

- Consider the following procedure, starting with some \( n \in \mathbb{N} \), where \( n > 0 \):
  - If \( n = 1 \), you are done.
  - If \( n \) is even, set \( n = n / 2 \).
  - Otherwise, set \( n = 3n + 1 \).
  - Repeat.

- **Question**: Given a natural number \( n > 0 \), does this process terminate?
If $n = 1$, stop.
If $n$ is even, set $n = n / 2$.
Otherwise, set $n = 3n + 1$.
Repeat.
The Hailstone Sequence

- Let $\Sigma = \{1\}$ and consider the language $L = \{1^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}$.
- Could we build a TM for $L$?
The Hailstone Turing Machine

- We can build a TM that works as follows:
  - If the input is $\varepsilon$, reject.
  - While the string is not $1$:
    - If the input has even length, halve the length of the string.
    - If the input has odd length, triple the length of the string and append a $1$.
  - Accept.
The Hailstone Turing Machine

If the input is $\varepsilon$, reject.

While the input is not $1$:

• If the input has even length, halve the length of the string.
• If the input has odd length, triple the length of the string and append a $1$.

Accept.
Does this Turing machine accept all nonempty strings?
The Collatz Conjecture

- It is *unknown* whether this process will terminate for all natural numbers.

- In other words, *no one knows whether the TM described in the previous slides will always stop running!*

- The conjecture (unproven claim) that this always terminates is called the Collatz Conjecture.
The Collatz Conjecture

“Mathematics may not be ready for such problems.” - Paul Erdős

- The fact that the Collatz Conjecture is unresolved is useful later on for building intuitions. Keep this in mind!
An Important Observation

-Unlike finite automata, which automatically halt after all the input is read, TMs keep running until they explicitly enter an accept or reject state.

-As a result, it’s possible for a TM to run forever without accepting or rejecting.

-This leads to several important questions:
  - How do we formally define what it means to build a TM for a language?
  - What implications does this have about problem-solving?
Very Important Terminology

- Let $M$ be a Turing machine.
- $M$ accepts a string $w$ if it enters an accept state when run on $w$.
- $M$ rejects a string $w$ if it enters a reject state when run on $w$.
- $M$ loops infinitely (or just loops) on a string $w$ if when run on $w$ it enters neither an accept nor a reject state.
- $M$ does not accept $w$ if it either rejects $w$ or loops infinitely on $w$.
- $M$ does not reject $w$ if it either accepts $w$ or loops on $w$.
- $M$ halts on $w$ if it accepts $w$ or rejects $w$. 

![Diagram showing states: Accept, Loop, Reject]
The Language of a TM

- The language of a Turing machine $M$, denoted $\mathcal{L}(M)$, is the set of all strings that $M$ accepts:
  \[
  \mathcal{L}(M) = \{ \, w \in \Sigma^* \mid M \text{ accepts } w \, \}
  \]

- For any $w \in \mathcal{L}(M)$, $M$ accepts $w$.
- For any $w \notin \mathcal{L}(M)$, $M$ does not accept $w$.
  - $M$ might reject, or it might loop forever.
- A language is called \textit{recognizable} if it is the language of some TM.
- A TM $M$ where $\mathcal{L}(M) = L$ is called a \textit{recognizer} for $L$.
- Notation: the class $\textit{RE}$ is the set of all recognizable languages.
  \[
  L \in \textit{RE} \iff L \text{ is recognizable}
  \]
What do you think? Does that correspond to what you think it means to solve a problem?
Deciders

- Some Turing machines always halt; they never go into an infinite loop.
- If $M$ is a TM and $M$ halts on every possible input, then we say that $M$ is a decider.
- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.
Decidable Languages

- A language $L$ is called **decidable** if there is a decider $M$ such that $\mathcal{L}(M) = L$.
- Equivalently, a language $L$ is decidable if there is a TM $M$ such that
  - If $w \in L$, then $M$ accepts $w$.
  - If $w \notin L$, then $M$ rejects $w$.
- The class $\mathbb{R}$ is the set of all decidable languages.

$L \in \mathbb{R} \leftrightarrow L$ is decidable
Next Time

• **Why Languages?**
  • Why do we use languages to model problem-solving?

• **Emergent Properties**
  • Larger phenomena made of smaller parts.

• **Universal Machines**
  • A single, “most powerful” computer.

• **Self-Reference**
  • Party tricks, and where recursion comes from.