Turing Machines
Part Three
Outline for Today

● **Why Languages and Strings?**
  ● We’ve been using languages to model problems. Why?

● **Universal Machines**
  ● A single computer that can compute anything computable anywhere.

● **Self-Referential Software**
  ● Programs that compute on themselves.
Recap from Last Time
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 Turing Machines

All Languages
Very Important Terminology

- Let $M$ be a Turing machine.
- $M$ accepts a string $w$ if it returns true on $w$.
- $M$ rejects a string $w$ if it returns false on $w$.
- $M$ loops infinitely (or just loops) on a string $w$ if when run on $w$ it neither returns true nor returns false.
- $M$ does not accept $w$ if it either rejects $w$ or loops 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 $w$. 

---

A diagram illustrating the concepts of accept, reject, loop, and halt.
Recognizers and Recognizability

- A TM $M$ is called a **recognizer** for a language $L$ over $\Sigma$ if the following statement is true:

  $\forall w \in \Sigma^*. (w \in L \leftrightarrow M \text{ accepts } w)$

- If you are absolutely certain that $w \in L$, then running a recognizer for $L$ on $w$ will (eventually) confirm this.
  - Eventually, $M$ will accept $w$.
- If you don’t know whether $w \in L$, running $M$ on $w$ may never tell you anything.
  - $M$ might loop on $w$ – but you can’t differentiate between “it’ll never give an answer” and “just wait a bit more.”
- This is a “weak” notion of “solving a problem.”
Deciders and Decidability

- A TM $M$ is called a **decider** for a language $L$ over $\Sigma$ if the following statements are true:

  $\forall w \in \Sigma^*. M$ halts on $w$. \\
  $\forall w \in \Sigma^*. (w \in L \iff M$ accepts $w)$

- In other words, $M$ accepts all strings in $L$ and rejects all strings not in $L$.

- In other words, $M$ is a recognizer for $L$, and $M$ halts on all inputs.

- If you aren’t sure whether $w \in L$, running $M$ on $w$ will (eventually) give you an answer to that question.

- This is a “strong” notion of solving a problem.
R and RE Languages

• The class $R$ consists of all decidable languages.

• The class $RE$ consists of all recognizable languages.

• By definition, we know $R \subseteq RE$.

• **Key Question:** Does $R = RE$?
New Stuff!
Strings, Languages, and Encodings
What problems can we solve with a computer?

What is a “problem?”
Decision Problems

• A **decision problem** is a type of problem where the goal is to provide a yes or no answer.

• Example: Bin Packing

  You're given a list of patients who need to be seen and how much time each one needs to be seen for. You're given a list of doctors and how much free time they have. Is there a way to schedule the patients so that they can all be seen?

• Example: Dominating Set Problem

  You're given a transportation grid and a number $k$. Is there a way to place emergency supplies in at most $k$ cities so that every city either has emergency supplies or is adjacent to a city that has emergency supplies?
A Model for Solving Problems

input

Computational Device

Yep

Nah
A Model for Solving Problems

input

Computational Device

Yep

Nah
A Model for Solving Problems

input

Computational Device

Yep

Nah
A Model for Solving Problems

input

Computational Device

Yep

Nah
A Model for Solving Problems

input → Turing Machine → Yep, Nah

input
A Model for Solving Problems

Turing Machine

input

Yep
(accept)

Nah
(reject)
A Model for Solving Problems

bool someFunctionName(string input) {

    // ... do something ...

}
A Model for Solving Problems

```c
bool isAnBn(string input) {
    // ... do something ...
}
```
A Model for Solving Problems

```cpp
bool isPalindrome(string input) {
    // ... do something ...
}
```
A Model for Solving Problems

input

Turing Machine

(accept)

Yep

(reject)

Nah

bool isLinkageGraph(Graph G) {
    // ... do something ...
}

How does this match our model?
A Model for Solving Problems

Turing Machine

input

Yep

Nah

(bool)

(accept)

(reject)

```c
bool containsCat(Picture P) {
    // ... do something ...
}
```
Humbling Thought:

*Everything on your computer is a string over \( \{0, 1\} \).*
Strings and Objects

• Think about how my computer encodes the image on the right.
• Internally, it's just a series of zeros and ones sitting on my hard drive.
Strings and Objects

- A different sequence of 0s and 1s gives rise to the image on the right.
- Every image can be encoded as a sequence of 0s and 1s, though not all sequences of 0s and 1s correspond to images.
Object Encodings

- If $Obj$ is some mathematical object that is *discrete* and *finite*, then we’ll use the notation $\langle Obj \rangle$ to refer to some way of encoding that object as a string.

- Think of $\langle Obj \rangle$ like a file on disk – it encodes some high-level object as a series of characters.

A few remarks about encodings:

- We don’t care how we encode the object, just that we can.
- The particular choice of alphabet isn’t important. Given any alphabet, we can always find a way of encoding things.
- We’ll assume we can perform “reasonable” operations on encoded objects.

$\langle \rangle = 110111001011\ldots$

$110$
Object Encodings

• If $Obj$ is some mathematical object that is *discrete* and *finite*, then we’ll use the notation $\langle Obj \rangle$ to refer to some way of encoding that object as a string.

• Think of $\langle Obj \rangle$ like a file on disk – it encodes some high-level object as a series of characters.

$$\langle \langle \rangle \rangle = 001101010001...$$

$$001$$
Object Encodings

• For the purposes of what we’re going to be doing, we aren’t going to worry about exactly how objects are encoded.

• For example, we can say \(\langle G \rangle\) to mean “some encoding of a graph \(G\)” without worrying about how it’s encoded.
  
  • Analogy: do you need to know how numbers are represented in Python to be a Python programmer? That’s more of a CS107 question.

• We’ll assume, whenever we’re dealing with encodings, that some person has figured out an encoding system for us and that we’re using that encoding system.
Object Encodings

- **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.

- Here are a bunch of different types of objects. Which of these objects can *always* be encoded as a string?
  - A DFA over the alphabet \{a, b\}.
  - A regular expression.
  - A subset of \{a, b\}*.  
  - A function $f$ from \{ $k \in \mathbb{N} \mid k < n$ \} to itself, for some $n \in \mathbb{N}$.
  - A graph whose nodes are the set \{ $k \in \mathbb{N} \mid k < n$ \}, for some $n \in \mathbb{N}$.  

A Model for Solving Problems

bool containsCat(Picture P) {
    // ... do something ...
}

Internally, this is a sequence of 0s and 1s.
A Model for Solving Problems

Turing Machine

input
(possibly encoded)

(accept)
Yep

(reject)
Nah

```
bool containsCat(Picture P) {
  // ... do something ...
}

Internally, this is a sequence of 0s and 1s.
```
bool containsCat(Picture P) {
    // ... do something ...
}
bool isLinkageGraph(Graph G) {
    // ... do something ... 
}
A Model for Solving Problems

```c
bool isDominatingSet(Graph G, Set D) {
    // ... do something ...
}
```
A Model for Solving Problems

input
(possibly encoded)

Turing Machine

(accept)
Yep

(reject)
Nah

```cpp
bool matchesRegex(string w, Regex R) {
    // ... do something ...
}
```

How does this match our model?
Encoding Groups of Objects

- Given a group of objects $Obj_1, Obj_2, \ldots, Obj_n$, we can create a single string encoding all these objects.
  - **Intuition 1:** Think of it like a .zip file, but without the compression.
  - **Intuition 2:** Think of it like a tuple or struct.

- We'll denote the encoding of all of these objects as a single string by $\langle Obj_1, \ldots, Obj_n \rangle$. 
A Model for Solving Problems

Turing Machine

input

Yep

Nah

(bool)

(accept)

(reject)

**bool** matchesRegex(string w, Regex R) {

// ... do something ...

}

These form one large bitstring.
A Model for Solving Problems

bool matchesRegex(string w, Regex R) {
    // ... do something ...
}

These form one large bitstring.
What problems can we solve with a computer?
Key Properties

- There are two key properties of computation that we will discuss:
  - *Universality*: There is a single computing device capable of performing any computation.
  - *Self-Reference*: Computing devices can ask questions about their own behavior.
- As you'll see, the combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.
Universal Machines
An Observation

• Think about how you interact with your physical computer.
  • You have a single, physical computer.
  • That computer then runs multiple programs.

• Contrast that with how we’ve worked with TMs.
  • We have a TM for \( \{ a^n b^n \mid n \in \mathbb{N} \} \). That TM will always perform that calculation and never do anything else.
  • We have a TM for the hailstone sequence. That TM can’t compose poetry, write music, etc.

• How do we reconcile this difference?
Can we make a “reprogrammable Turing machine?”
A TM Simulator

- It is possible to program a TM simulator on an unbounded-memory computer.
  - You’ve seen this in class, and you’ll use one on PS8.
- We could imagine it as a method
  
  \[
  \text{bool simulateTM(TM M, string w)}
  \]

  with the following behavior:
  - If \( M \) accepts \( w \), then simulateTM(M, w) returns \text{true}.
  - If \( M \) rejects \( w \), then simulateTM(M, w) returns \text{false}.
  - If \( M \) loops on \( w \), then simulateTM(M, w) loops infinitely.
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?

\[
\begin{align*}
M & \quad \text{Auk:} \\
& \quad \quad \text{Move Left} \\
& \quad \quad \text{Write 'k'} \\
& \quad \quad \text{Goto Moa} \\
& \quad \quad \ldots
\end{align*}
\]

...input...

\[
\begin{align*}
\text{simulateTM} & \quad \text{true!} \\
& \quad \text{(loop)} \\
& \quad \text{false!}
\end{align*}
\]
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?
The Universal Turing Machine

**Theorem (Turing, 1936):** There is a Turing machine $U_{TM}$ called the universal Turing machine that, when run on an input of the form $⟨M, w⟩$, where $M$ is a Turing machine and $w$ is a string, simulates $M$ running on $w$ and does whatever $M$ does on $w$ (accepts, rejects, or loops).

The observable behavior of $U_{TM}$ is the following:

- If $M$ accepts $w$, then $U_{TM}$ accepts $⟨M, w⟩$.
- If $M$ rejects $w$, then $U_{TM}$ rejects $⟨M, w⟩$.
- If $M$ loops on $w$, then $U_{TM}$ loops on $⟨M, w⟩$.

$U_{TM}$ does on $⟨M, w⟩$ what $M$ does on $w$. 

---

**Auk:**
- Move Left
- Write 'k'
- Goto Moa

**Tern:**
- If Blank Goto Heron
- Write 'q'
- Move Right
- ...

**Universal TM**

...input...

...
$U_{\text{TM}}$ as a Recognizer

- $U_{\text{TM}}$, when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, will
  
  ... accept $\langle M, w \rangle$ if $M$ accepts $w$,
  
  ... reject $\langle M, w \rangle$ if $M$ rejects $w$, and
  
  ... loop on $\langle M, w \rangle$ if $M$ loops on $w$.

- Although we didn’t design $U_{\text{TM}}$ as a recognizer, it does recognize some language.

- Which language is that?
$U_{TM}$ as a Recognizer

- $U_{TM}$, when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, will
  
  ... accept $\langle M, w \rangle$ if $M$ accepts $w$,
  
  ... reject $\langle M, w \rangle$ if $M$ rejects $w$, and
  
  ... loop on $\langle M, w \rangle$ if $M$ loops on $w$.

- Let’s let $A_{TM}$ be the language recognized by the universal TM $U_{TM}$. This means that

  $\forall M. \forall w \in \Sigma^*. (U_{TM} \text{ accepts } \langle M, w \rangle \leftrightarrow \langle M, w \rangle \in A_{TM})$
U_TM as a Recognizer

- U_TM, when run on a string ⟨M, w⟩, where M is a TM and w is a string, will
  ... accept ⟨M, w⟩ if M accepts w,
  ... reject ⟨M, w⟩ if M rejects w, and
  ... loop on ⟨M, w⟩ if M loops on w.

- Let’s let A_{TM} be the language recognized by the universal TM U_TM. This means that
  \[ \forall M. \forall w \in \Sigma^*. (M \text{ accepts } w \iff ⟨M, w⟩ \in A_{TM}) \]

- So we have
  \[ A_{TM} = \{ ⟨M, w⟩ \mid M \text{ is a TM and } M \text{ accepts } w \} \]
The Language $A_{TM}$

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

- Here’s a complicated expression. Can you simplify it?
  $$\langle U_{TM}, \langle M, w \rangle \rangle \in A_{TM}.$$  
- Given the definition of $A_{TM}$ and $U_{TM}$, the following statements are all equivalent to one another.
  - $M$ accepts $w$.
  - $U_{TM}$ accepts $\langle M, w \rangle$.
  - $\langle M, w \rangle \in A_{TM}$. 
Uh... so what?
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the computer with a TM. (This gave us the universal TM.)
- What happens if we replace the TM with a computer program?

```
M
Auk:
  Move Left
  Write 'k'
  Goto Moa
...

w
...input...
```

```
simulateTM
```

```
true!
(loop)
false!
```
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the computer with a TM. (This gave us the universal TM.)
- What happens if we replace the TM with a computer program?

```
for (int i = 2; i < n; i++) {
    if (n % i == 0) {
        ...input...
        simulateProgram
    }
}
```
Why Does This Matter?

• We now have a computer program that runs other computer programs!
  
• An **interpreter** is a program that simulates other programs. Python programs are usually executed by interpreters. Your web browser interprets JavaScript code when it visits websites.

• A **virtual machine** is a program that simulates an entire operating system. Virtual machines are used in computer security, cloud computing, and even by individual end users.

• It’s not a coincidence that this is possible – Turing’s 1936 paper says that any general-purpose computing system must be able to do this!

```java
for (int i = 2; i < n; i++) {
    if (n % i == 0) …
}
```
Why Does This Matter?

- The key idea behind the universal TM is that TMs can be fed as inputs into other TMs.
  - Similarly, an interpreter is a program that takes other programs as inputs.
  - Similarly, an emulator is a program that takes entire computers as inputs.
- This hits at the core idea that computing devices can perform computations on other computing devices.
Self-Reference
Self-Referential Programs

• If TMs can take other TMs as input, can they take themselves as input??
  
  \textbf{YES.}

• TMs can take their own code as input, and ask questions about (or even execute!) their own code.

• In fact, any computing system that’s equal to a Turing machine in power possesses some mechanism for self-reference!

• Want to see how deep the rabbit hole goes? Take CS154!
Writing Self-Referential Code!

*See zip file with lecture slides for code.*
Quines

- A **Quine** is a special kind of self-referential program that, when run, prints its own source code.
- Believe it or not, it’s possible to write such a program!
Self-Referential Programs

- **Claim**: Going forward, assume that any function has the ability to get access to its own source code.
- This means we can write programs like the ones shown here:

```cpp
bool equalsMe(string input) {
    string me = mySource();
    return input == me;
}
```
Next Time

- **Self-Defeating Objects**
  - Objects “too powerful” to exist.

- **Undecidable Problems**
  - Problems truly beyond the limits of algorithmic problem-solving!

- **Consequences of Undecidability**
  - Why does any of this matter outside of Theoryland?