Turing Machines
Part Two
Outline for Today

- **The Church-Turing Thesis**
  - How powerful are Turing machines?
- **Decidability and Recognizability**
  - Two notions of “solving a problem.”
- **Universal Machines**
  - A single computer that can compute anything computable anywhere.
- **Self-Referential Software**
  - Programs that compute on themselves.
Just how powerful are Turing machines?
A Leap of Faith

• **Claim:** A TM is powerful enough to simulate any computer program that gets an input, processes that input, then returns some result.

• 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.
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
Problems solvable by any feasible computing machine.

Regular Languages

All Languages
Problems solvable by Turing Machines

Regular Languages

All Languages
TMs and Computation

- 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!
Decidability and Recognizability
What problems can we solve with a computer?

What kind of 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

- 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.
- Does the Hailstone Sequence terminate for...
  - $n = 5$?
  - $n = 20$?
  - $n = 7$?
  - $n = 27$?
The Hailstone Sequence

• Let $\Sigma = \{ a \}$ and consider the language
  
  $L = \{ a^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}$. 

• 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 a:
    - 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 a.
  - 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 the hailstone sequence always terminates is called the **Collatz Conjecture**.
• This problem has eluded a solution for a long time. The influential mathematician Paul Erdős is reported to have said “Mathematics may not be ready for such problems.”
An Important Observation

• Unlike finite automata, which automatically halt after all the input is read, TMs keep running until they explicitly return true or return false.

• 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 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 $w$.
Recognizers and Recognizability

- A TM $M$ is called a **recognizer** for a language $L$ over $\Sigma$ if the following statement is true:
  \[ \forall w \in \Sigma^*. \ (w \in L \iff M \text{ accepts } w) \]

- 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.”

- Does that feel like “solving a problem” to you?
Recognizers and Recognizability

- The hailstone TM $M$ we saw earlier is a recognizer for the language

$$L = \{ a^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}.$$

- If the sequence does terminate starting at $n$, then $M$ accepts $a^n$.

- If the sequence doesn’t terminate, then $M$ loops forever on $a^n$ and never gives an answer.

- If you somehow knew the hailstone sequence terminated for $n$, this machine would (eventually) confirm this. If you didn’t know, this machine might not tell you anything.
Recognizers and Recognizability

- The class \( \text{RE} \) consists of all recognizable languages.
- Formally speaking:
  \[
  \text{RE} = \{ L \mid L \text{ is a language and there's a recognizer for } L \}
  \]
- You can think of \( \text{RE} \) as “all problems with yes/no answers where “yes” answers can be confirmed by a computer.”
  - Given a recognizable language \( L \) and a string \( w \in L \), running a recognizer for \( L \) on \( w \) will eventually confirm \( w \in L \).
  - The recognizer will never have a “false positive” of saying that a string is in \( L \) when it isn’t.
- This is a “weak” notion of solving a problem.
- Is there a “stronger” one?
Deciders and Decidability

- Some, but not all, TMs have the following property: the TM halts on all inputs.
- If you are given a TM $M$ that always halts, then for the TM $M$, the statement “$M$ does not accept $w$” means “$M$ rejects $w$.”
Deciders and Decidability

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

$$\forall w \in \Sigma^*. \ M \text{ halts on } w.$$ 

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

- 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.
Deciders and Decidability

• The hailstone TM $M$ we saw earlier is a **recognizer** for the language

\[ L = \{ \ a^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \ \} . \]

• If the hailstone sequence terminates for $n$, then $M$ accepts $a^n$. If it doesn’t, then $M$ does not accept $a^n$.

• We honestly don’t know if $M$ is a decider for this language.
  • If the hailstone sequence always terminates, then $M$ always halts and is a decider for $L$.
  • If the hailstone sequence doesn’t always terminate, then $M$ will loop on some inputs and isn’t a decider for $L$. 
Deciders and Decidability

- The class $\mathbf{R}$ consists of all decidable languages.
- Formally speaking:
  \[ \mathbf{R} = \{ L \mid L \text{ is a language and there’s a decider for } L \} \]
- You can think of $\mathbf{R}$ as “all problems with yes/no answers that can be fully solved by computers.”
  - Given a decidable language, run a decider for $L$ and see what happens.
  - Think of this as “knowledge creation” – if you don’t know whether a string is in $L$, running the decider will, given enough time, tell you.
- The class $\mathbf{R}$ contains all the regular languages, all the context-free languages, most of CS161, etc.
- This is a “strong” notion of solving a problem.
R and RE Languages

• Every decider for $L$ is also a recognizer for $L$.
• This means that $R \subseteq RE$.
• Hugely important theoretical question:

$$R \equiv RE$$

• That is, if you can just confirm “yes” answers to a problem, can you necessarily solve that problem?
Which Picture is Correct?

Regular Languages

R

RE

All Languages
Which Picture is Correct?

- Regular Languages
- R
- RE
- All Languages
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

Computational Device

input

Yep

Nah
A Model for Solving Problems

input

Turing Machine

Yep

Nah
A Model for Solving Problems

input

Turing Machine

(accept)

Yep

(reject)

Nah
A Model for Solving Problems

bool someFunctionName(string input) {
    // … do something …
}

Turing Machine
A Model for Solving Problems

bool isAnBn(string input) {
    // ... do something ...
}

Turing Machine

input

(accept)

Yep

(reject)

Nah
A Model for Solving Problems

bool isPalindrome(string input) {
    // ... do something ...
}

input → Turing Machine

(accept)

Yep

(reject)

Nah
A Model for Solving Problems

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

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

**bool containsCat(Picture P) {**

```
// ... do something ...
```

**}**

How does this match our model?
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.

$\langle \rangle = 110111001011...110$
Object Encodings

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

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

\[ \langle \text{Obj} \rangle = 001101010001\ldots001 \]
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 ⟨137⟩ to mean “some encoding of 137” 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 Smart, Attractive, Witty person has figured out an encoding system for us and that we’re using that encoding system.
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

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

Internally, this is a sequence of 0s and 1s.
bool containsCat(Picture P) {
    // ... do something ...
}

A Model for Solving Problems

Turing Machine

input

(possibly encoded)

(accept)

Yep

(reject)

Nah
A Model for Solving Problems

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

Turing Machine

input

(possibly encoded)

(accept)

Yep

(reject)

Nah
A Model for Solving Problems

bool isDominatingSet(Graph G, Set D) {
    // ... do something ...
}

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

input
(possibly encoded)

Turing Machine

(accept)
Yep

(reject)
Nah

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

input → Turing Machine

(accept)

Yep

Nah

(reject)

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

These form one large bitstring.
bool matchesRegex(string w, Regex R) {
    // ... do something ...
}

These form one large bitstring.
What problems can we solve with a computer?
Time-Out for Announcements!
Problem Set Five Graded

- Your diligent and hardworking TAs have finished grading PS4. Grades and feedback are now available on Gradescope.

![](image)

- As always, please review your feedback! Knowing where to improve is more important than just seeing a raw score.

- Did we make a mistake? Regrades on Gradescope will open tomorrow and are due in one week.
Problem Sets

• Problem Set 6 was due today at 2:30 PM.
• Problem Set 7 (our final problem set!) is out now and will be due Wednesday, August 10th at 2:30 PM.
  • This assignment will be shorter since you don’t have a full week to work on it.
Final Exam

• The final goes out on Wednesday, August 10th at 2:30PM Pacific. It comes due on Friday, August 12th at 7:00PM Pacific.
  • The exam must be completed individually.
  • It’s open-book, open-note, and closed-other-humans.
  • It covers PS1 – PS7 and weeks 1 to 7 of lectures. Note: week 8 lectures will not be tested.
• Extra practice problems are up on the course website!
Back to CS103!
Emergent Properties
Emergent Properties

• An emergent property of a system is a property that arises out of smaller pieces that doesn't seem to exist in any of the individual pieces.

• Examples:
  • Individual neurons work by firing in response to particular combinations of inputs. Somehow, this leads to consciousness, love, and ennui.
  • Individual atoms obey the laws of quantum mechanics and just interact with other atoms. Somehow, it's possible to combine them together to make iPhones and pumpkin pie.
Emergent Properties of Computation

• All computing systems equal to Turing machines exhibit several surprising emergent properties.

• If we believe the Church-Turing thesis, these emergent properties are, in a sense, “inherent” to computation. Computation can’t exist without them.

• These emergent properties are what ultimately make computation so interesting and so powerful.

• As we'll see, though, they're also computation's Achilles heel – they're how we find concrete examples of impossible problems.
Two Emergent Properties

• There are two key emergent 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.
- We could imagine it as a method
  
  ```
  bool simulateTM(TM M, string w)
  ```

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

\[ \text{simulateTM} \]
\[ \text{true!} \rightarrow \text{(loop)} \]
\[ \text{false!} \]
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.

\[ M \]

\[ w \]

...input...

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

simulateTM

true!

false!

(loop)
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?

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

\[ w \quad \ldots\text{input}\ldots \]

\[ \text{Tern:} \quad \begin{align*}
& \text{If Blank Goto Heron} \\
& \text{Write 'q'} \\
& \text{Move Right} \\
& \ldots
\end{align*} \]

\[ \text{TM that runs other TMs} \]

\[ \text{accept!} \quad \text{(loop)} \quad \text{reject!} \]
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 $⟨M, w⟩$.

---

**Universal TM**

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

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

- **U_{TM} does to ⟨M, w⟩**
  - what
  - **M does to w.**  

- **Accept!**
- **Reject!**  
- **(loop)**
$U_{TM}$ 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 $w$.

- Although we didn’t design $U_{TM}$ 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
  $\forall x \in \Sigma^*. (U_{TM} \text{ accepts } x \leftrightarrow x \in A_{TM})$
\textbf{U}_\text{TM} \text{ 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 \(w\).

- Let’s let \(A_{\text{TM}}\) be the language recognized by the universal TM \(U_{\text{TM}}\). This means that
  \[\forall M. \forall w \in \Sigma^*. (U_{\text{TM}} \text{ accepts } \langle M, w \rangle \leftrightarrow \langle M, w \rangle \in A_{\text{TM}})\]
$U_{\text{TM}}$ 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 $w$.
- Let’s let $A_{\text{TM}}$ be the language recognized by the universal TM $U_{\text{TM}}$. This means that
  \[
  \forall M. \forall w \in \Sigma^*. (M \text{ accepts } w \iff \langle M, w \rangle \in A_{\text{TM}})
  \]
- So we have
  \[
  A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}
  \]
The Language $A_{\text{TM}}$

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

- Here’s a complicated expression. Can you simplify it?
  \[
  \langle U_{\text{TM}}, \langle M, w \rangle \rangle \in A_{\text{TM}}.
  \]

- Given the definition of $A_{\text{TM}}$ and $U_{\text{TM}}$, the following statements are all equivalent to one another.
  - $M$ accepts $w$.
  - $U_{\text{TM}}$ accepts $\langle M, w \rangle$.
  - $\langle M, w \rangle \in A_{\text{TM}}$. 
Regular Languages

All Languages

\( A_{\text{TM}} \)

RE
Uh... so what?
Reason 1: *It has practical consequences.*
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?
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...
```
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!

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

• The key idea behind the universal TM is that idea 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.
Reason 2: *It’s philosophically interesting.*
Can Computers Think?

• On May 15, 1951, Alan Turing delivered a radio lecture on the BBC on the topic of whether computers can think.

• He had the following to say about whether a computer can be thought of as an electric brain...
“In fact I think [computers] could be used in such a manner that they could be appropriately described as brains. I should also say that

‘If any machine can be appropriately described as a brain, then any digital computer can be so described.’

This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

It can be shown to follow from a characteristic property of digital computers, which I will call their universality. A digital computer is a universal machine in the sense that it can be made to replace any machine of a certain very wide class. It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results. In order to arrange for our computer to imitate a given machine it is only necessary to programme the computer to calculate what the machine in question would do under given circumstances, and in particular what answers it would print out. The computer can then be made to print out the same answers.

If now some machine can be described as a brain we have only to programme our digital computer to imitate it and it will also be a brain.”
Self-Referential Software
Self-Referential Programs

• If TMs can take other TMs as input, could they take themselves as input?

  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 in power to a Turing machine possesses some mechanism for self-reference.

• Want to see how deep the rabbit hole goes? Take CS154!
Quines

- A **Quine** is a special kind of self-referential program that, when run, prints its own source code.
- Believe it or not, it is possible to write such a program!
- *See zip file with lecture slides for code.*
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 one shown here:

```cpp
bool narcissist(string input) {
    string me = /* source code of narcissist */;
    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?