At the Foot of the Mountains of Undecidability

16 November 2023
At the Foot of the Mountains of Undecidability
“What problems can we solve with a computer?”
“What problems can we solve with a computer?”
The Church–Turing Thesis claims that:

Every effective method of computation is either equivalent to or weaker than a Turing machine.
Regular languages \(\subseteq\) CFLs

Problems solvable by any feasible computing machine

All languages
All languages

Regular languages

CFLs

Problems solvable by Turing machines

All languages
Because of the Church–Turing Thesis, we can start to be less detailed with our TM descriptions.
A *high-level description* of a Turing machine is a description like this:

\[ M = \text{“On input } x:\]
Repeat the following:
\[ \text{If } |x| \leq 1, \text{ accept.} \]
\[ \text{If the first and last symbols of } x \text{ aren’t the same, reject.} \]
\[ \text{Remove the first and last characters of } x.” \]

High-level descriptions are just a kind of pseudocode!
def M(x: str) -> bool:
    while True:
        if len(x) <= 1:
            return True
        if x[0] != x[-1]
            return False
        x = x[1:-1]

M = "On input x:
    Repeat the following:
    If |x| ≤ 1, accept.
    If the first and last symbols of x aren’t the same, reject.
    Remove the first and last characters of x."
Every Turing machine
resembles some input,
do some work, then
(optionally) accepts or rejects.

So, we can model a Turing machine as a computer
program where
the program’s logic is written in a normal programming language, and
the program (optionally) returns True to immediately accept the input
or returns False to immediately reject the input.
“What problems can we solve with a computer?”

What does it mean to “solve” a problem?
Unlike finite automata, which automatically halt after reading the input, Turing machines keep running until they explicitly enter an accept or reject state.

As such, it’s possible for a Turing machine to run forever without accepting or rejecting.
Accept

Does not reject

Does not accept

Reject

Loop

Halts
Recognizers and recognizability

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

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

This is a weak notion of “solving a problem”:

If you are absolutely certain that $w \in L$, then running a recognizer for $L$ on $w$ will 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 longer”!
Deciders and decidability

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

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

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

This is a strong notion of “solving a problem”:

If you don’t know whether $w \in L$, running $M$ on $w$ will (eventually) give you an answer to that question.
**R and RE languages**

The class **R** consists of all decidable languages.

The class **RE** consists of all recognizable languages.
A feel for R and RE

You want to see if the hailstone sequence terminates for some $n \in \mathbb{N}$.

An RE perspective: Run the hailstone sequence starting at $n$. If it stops, return true. But if the hailstone sequence doesn’t terminate, you’ll never learn this.

An R perspective: Perform some calculation on the number $n$ that determines whether the hailstone sequence terminates, but without actually running the hailstone sequence.
A feel for R and RE

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$. 
A feel for $R$ and $RE$

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

Not whether the language of the DFA is $a^n b^n$, which we proved is impossible, just whether it accepts any string of this form!
A feel for R and RE

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

An RE perspective: Run the DFA on $a^0 b^0$, $a^1 b^1$, $a^2 b^2$, etc. If the DFA ever accepts, return true. But, if not, you may never learn this.

An R perspective: Look at the structure of the DFA and, somehow, determine whether it accepts any strings of this form, but without running the DFA on all of them.
A feel for R and RE

Say you’re working on a CS assignment. You wonder if there’s any input that will make your program crash.

An RE perspective: Try running the program on every possible input. If you see it crash, return true. If it never crashes, you will never learn this.

An R perspective: Look at the source code and somehow determine, with 100% certainty, whether the program will ever crash.
A feel for R and RE

You have an X. You want to see if there’s a Y where X and Y go well together.

An RE perspective: List all the Ys in some order and check if X and Y go well together. If so, return true. If not, you might not learn anything.

An R perspective: Look at X and, somehow, determine whether such a Y exists without checking all Ys.
**Intuition 1:** Problems in **RE** are ones that can be approached by doing some sort of exhaustive search over a potentially infinite list of options.

**Intuition 2:** Problems in **R** are ones that can be solved without having to exhaustively try infinitely many possibilities.
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 \( R = RE \)?
Is this right?

- Regular languages
- CFLs
- RE
- R

All languages
Or this?

- Regular languages
- CFLs
- \( R \)
- \( RE \)

All languages
“What problems can we solve with a computer?”

What is a “problem”?
A decision problem is a type of problem where the goal is to answer yes or no.

Example: Bin Packing

You’re given a list of patients who need to be seen and how much time each 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: Route Planning

You’re given a transportation grid of a city, a start location, a destination location, and information about the traffic over the course of the day. Given a time limit $T$, is there a way to drive from the start location to the end location in at most $T$ hours?
Computational device

input

Yes

No
def some_function_name(input: str) -> bool:
    ...

input → Turing machine → Accept

input → Turing machine → Reject
def is_an_bn(input: str) -> bool:
   ...

input → Turing machine → Accept

input → Turing machine → Reject
def is_palindrome(input: str) -> bool:
    ...

Turing machine

input

Accept

Reject
def is_fully_connected(g: Graph) -> bool:
...

How does this match our model?
def contains_cat(i: Image) -> bool:
...

How does this match our model?
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 on my hard drive.

Ziggy, an exemplary cat
Strings and objects

A different sequence of zeros and ones gives rise to the image on the right.

Every image can be encoded as a sequence of zeros and ones – though not all sequences of zeros and ones correspond to images!
If $Obj$ is a discrete, finite mathematical object, then we’ll use the notation $\langle Obj \rangle$ to refer to some reasonable encoding of that object as a string of characters.

$$\langle \text{cat} \rangle = 11001101000101110100101 \ldots$$
If $Obj$ is a discrete, finite mathematical object, then we’ll use the notation $\langle Obj \rangle$ to refer to some reasonable encoding of that object as a string of characters.

$\langle \rangle = 000100110100111000010110 \ldots$
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.

Generally we’ll assume that some clever person has already figured out a way to encode what we want.

For example, we can just say, e.g., ⟨137⟩ to mean “some encoding of 137” without worrying about how it’s encoded.
Great intuition: If you can store an object as a file on disk, then you can encode it as a string.
Here are some types of objects:

- A DFA over the alphabet \{a, b\}
- A regular expression
- A subset of \{a, b\}*
- A graph whose nodes are the set \{k \in \mathbb{N} | k < n\}, for some \(n \in \mathbb{N}\).

Which of these can *always* be encoded as a string?
Here are some types of objects:

- A DFA over the alphabet \{a, b\}
- A regular expression
- A subset of \{a, b\}^*
- A graph whose nodes are the set \{k \in \mathbb{N} \mid k < n\}, for some \(n \in \mathbb{N}\).

Which of these can \textit{always} be encoded as a string?
Here are some types of objects:

- A DFA over the alphabet \{a, b\}
- A regular expression ✓
- A subset of \{a, b\}*
- A graph whose nodes are the set \( \{ k \in \mathbb{N} \mid k < n \} \), for some \( n \in \mathbb{N} \).

Which of these can always be encoded as a string?

It’s already a string!
Here are some types of objects:

- A DFA over the alphabet \{a, b\}
- A regular expression
- A subset of \{a, b\}^*
- A graph whose nodes are the set \{k \in \mathbb{N} \mid k < n\}, for some \(n \in \mathbb{N}\).

Which of these can \textit{always} be encoded as a string?
Here are some types of objects:

A DFA over the alphabet \{a, b\}

A regular expression

A subset of \{a, b\}^* \times

A graph whose nodes are the set \{k \in \mathbb{N} \mid k < n\} , for some \(n \in \mathbb{N}\).

Which of these can \textit{always} be encoded as a string?

\textbf{The subset might be infinite, and there’s no guarantee that it has a finite description.}
Here are some types of objects:

- A DFA over the alphabet \( \{a, b\} \)
- A regular expression
- A subset of \( \{a, b\}^* \)
  - A graph whose nodes are the set \( \{k \in \mathbb{N} \mid k < n\} \), for some \( n \in \mathbb{N} \). ✓

Which of these can *always* be encoded as a string?
def contains_cat(i: Image) -> bool:
...

Internally, this is a sequence of 0s and 1s
def contains_cat(i: Image) -> bool:
    ...

Internally, this is a sequence of 0s and 1s
def contains_cat(i: Image) -> bool:
    ...

Turing machine

input

(possibly encoded)

Accept

Reject
def is_fully_connected(g: Graph) -> bool:
    ...

Turing machine

input (possibly encoded)

Accept

Reject
def is_dominating_set(g: Graph, d: set) -> bool:
    ...

How does this match our model?
How does this match our model?
Encoding groups of objects

Given a finite 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 without the compression.

*Intuition 2:* Think of it like a tuple.

We can denote the encoding of all of these objects as a single string by \( \langle Obj_1, Obj_2, \ldots, Obj_n \rangle \).
def matches_regex(s: str, r: RegEx) -> bool:
    ...

These form one large bit-string.
def matches_regex(s: str, r: RegEx) -> bool:
    ...

These form one large bit-string.
Our goal is to speak of *computers solving problems*. We model this by looking at *Turing machines recognizing languages*. By turning any problem into an equivalent *decision problem*, this precisely captures what we’re interested in.
“What problems can we solve with a computer?”
We're getting closer!

However, to understand the answer, we're going to need to step back for a moment.
Let’s think about *emergent properties*.

An emergent property of a system is a property that arises out of smaller pieces but which doesn’t seem to exist in any of the individual pieces, e.g.,

Individual neurons fire in response to particular combinations of inputs and this gives rise to human consciousness.

Individual atoms obey the laws of quantum mechanics and just interact with other atoms, and this gives rise to literally everything.
All computing systems equal to Turing machines exhibit several surprising emergent properties.

If we believe the Church–Turing Thesis, these must be inherent to computation; computation can’t exist without them.

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

But we’ll see they’re also computation’s Achilles heel – they’re how we find concrete examples of impossible problems.
The key emergent properties of computation that we’ll discuss are:

*Universality:* There is a single computing device capable of performing any computation.

*Self-reference:* Computing devices can ask questions about their own behavior.

We’ll see that the combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.
Universality
We’ve been designing Turing machines to solve specific problems.

Do you have a dedicated computer for each task you need to perform?

- Your email computer?
- Your word-processing computer?
- Your cute-cat-picture computer?
Can we make a “reprogrammable Turing machine”?
A Turing machine simulator

It’s possible to program a Turing machine simulator on an unbounded memory computer.

If we accept some limits on the “infinite” tape, we can even do this on a real computer.
Turing’s World 3.0, 1993
A Turing machine simulator

While a simulator like this is an interactive tool to help us understand the theoretical model, we can also imagine it as a procedure

\[
\text{simulate_tm}(M: \text{TM}, \ w: \text{str}) \rightarrow \text{bool}
\]

with the following behavior:

If \( M \) accepts \( w \), then \( \text{simulate_tm}(M, \ w) \) returns True.
If \( M \) rejects \( w \), then \( \text{simulate_tm}(M, \ w) \) returns False.
If \( M \) loops on \( w \), then \( \text{simulate_tm}(M, \ w) \) loops infinitely.
simulate_tm

true!

false!

(loop)
Anything that can be done with an unbounded-memory computer can be done with a Turing machine.

So there must be a Turing machine that has the behavior of the `simulate_tm` method.
TM that runs other TMs

accept!

(loop)

reject!
M

...input...

Universal TM

accept!

(loop)

reject!
THEOREM (Turing, 1936) There is a Turing machine $U$ called the universal Turing machine that, when run on an input of the form $\langle M, w \rangle$, where $M$ is a Turing machine and $w$ is a string, simulates $M$ running on $w$ and does whatever $M$ does on $w$.

If $M$ accepts $w$, then $U$ accepts $\langle M, w \rangle$.

If $M$ rejects $w$, then $U$ rejects $\langle M, w \rangle$.

If $M$ loops on $w$, then $U$ loops on $\langle M, w \rangle$. 
$U$ does to input $\langle M, w \rangle$ what $M$ does on input $w$. 
The universal Turing machine $U$, schematically
Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

**Machine $M$**

![Diagram of a DFA with states $q_0$, $q_1$, $q_{\text{acc}}$, and $q_{\text{rej}}$. The transitions include $\square \rightarrow \square$, $a \rightarrow \square$, and $\square \rightarrow \square$.]

**Input $w$**

```
... a a a a ...```

Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

**Machine $M$**

![Diagram of a DFA with states $q_0$, $q_1$, $q_{\text{acc}}$, and $q_{\text{rej}}$. The transitions include $\square \rightarrow \square$, $a \rightarrow \square$, and $\square \rightarrow \square$.]

**Input $w$**

```
... a a a a ...```
Machine $M$

Take $M$ and write it down as a string.
(Remember how we can encode the finite-state control as table.)

Input $w$

\[
\begin{array}{ccccccc}
\vdots & a & a & a & a & a & \vdots \\
\end{array}
\]
Take $M$ and write it down as a string. 
(Remember how we can encode the finite-state control as table.)

Machine $M$

Input $w$

$$\vdots \quad a \quad a \quad a \quad a \quad \vdots$$

$$\vdots \quad q_0 \quad a \quad \square \quad R \quad \vdots \quad q_1 \quad a \quad \vdots \quad \vdots$$

$$M$$
Now take your input $w$ and write it down too.
Now take your input $w$ and write it down too.
Feed this into $U$. 

**Machine $M$**

- **Start State**: $q_0$
- **Accepting State**: $q_{acc}$
- **Rejecting State**: $q_{rej}$
- Transitions:
  - $a \rightarrow \square$, $R$
  - $\square \rightarrow \square$, $R$

**Input $w$**

```
... a a a a ...
```

**Input $\langle M, w \rangle$**

```
... q_0 a \square R ... q_1 a ... a a a a ...
```

$M$ and $w$ are concatenated.
Machine $M$

- **Start state**: $q_0$
- **Accepting state**: $q_{\text{acc}}$
- **Rejecting state**: $q_{\text{rej}}$

- Transition diagram:
  - $\square \rightarrow \square, R$
  - $a \rightarrow \square, R$

Input $w$:

```
⋯ a a a a a ⋯
```

Universal TM $U$

- **Start state**: $q_0$
- **Accepting state**: $q_{\text{acc}}$
- **Rejecting state**: $q_{\text{rej}}$

- Transition logic:
  - Look at next char of $w$
  - If $M$ is in accepting state
  - If $M$ is in rejecting state
  - Look up what $M$ should do upon reading $w$
  - Update state and tape

Input $\langle M, w \rangle$:

```
⋯ q_0 a □ R ⋯ q_1 a ⋯ a a a a a ⋯
```

- $M$:
  - Machine $M$.
- $w$:
  - Input string $w$. 

- $\langle M, w \rangle$:
  - Input pair $\langle M, w \rangle$. 

- Transition logic:
  - Look at next char of $w$
  - If $M$ is in accepting state
  - If $M$ is in rejecting state
  - Look up what $M$ should do upon reading $w$
  - Update state and tape

- Output:
  - Accepting state $q_{\text{acc}}$
  - Rejecting state $q_{\text{rej}}$
Machine $M$

Universal TM $U$

Input $w$

Input $\langle M, w \rangle$
Machine $M$

- **Start State**: $q_0$
- **Accepting State**: $q_{acc}$
- **Rejecting State**: $q_{rej}$

- **Transition**: $a \rightarrow \square, R$

Universal TM $U$

- **Start State**: $q_0$
- **Accepting State**: $q_{acc}$
- **Rejecting State**: $q_{rej}$

- **Transition**: $a \rightarrow \square, R$

Input $w$

- $\ldots \ a \ a \ a \ a \ a \ \ldots$

Input $\langle M, w \rangle$

- $\ldots q_0 \ a \ \square \ R \ \ldots q_1 \ a \ \ldots \ a \ a \ a \ a \ a \ \ldots$

- $M$
- $w$
Machine $M$

Start

$q_0 \xrightarrow{a} □, R \quad □ \rightarrow □, R \quad q_{\text{rej}}$

$q_1 \xrightarrow{a} □, R \quad □ \rightarrow □, R \quad q_{\text{rej}}$

Input $w$

$\cdots a a a a a \cdots$

Universal TM $U$

Start

$q_{\text{acc}}$

Look at next char of $w$

If $M$ is in accepting state

$q_{\text{acc}}$

Look up what $M$ should do upon reading $w$

If $M$ is in rejecting state

$q_{\text{rej}}$

Update state and tape

Input $⟨M, w⟩$

$\cdots q_0 a □ R \cdots q_1 a a a a a \cdots$

$M$

$w$
Machine $M$

- $\text{start} \rightarrow q_0$
- $a \rightarrow q_1$,
- $\square \rightarrow \square$, $R$
- $q_1 \rightarrow q_{\text{rej}}$
- $\square \rightarrow \square$, $R$

Universal TM $U$

- $\text{start} \rightarrow q_{\text{acc}}$
- Look at next char of $w$ of $\langle M, w \rangle$
- Look up what $M$ should do upon reading $w$
- If $M$ is in accepting state
- If $M$ is in rejecting state
- Update state and tape

Input $w$

- $\ldots a a a a \ldots$

Input $\langle M, w \rangle$

- $\ldots q_0 a \square R \ldots q_1 a \ldots a a a a \ldots$

- $M$
- $w$
Universal TM $U$

If $M$ is in accepting state

*Look at next char of $w$*

If $M$ is in rejecting state

*Look up what $M$ should do upon reading $w$*

Update state and tape

Input $\langle M, w \rangle$

\[
\ldots q_0 a \square R \ldots q_1 a \ldots a a a a \ldots
\]
Machine $M$

- **start**
  - **$q_0$**
    - $\square \rightarrow \square, R$
    - $a \rightarrow \square, R$
  - **$q_1$**
    - $a \rightarrow \square, R$
  - **$q_{\text{rej}}$**
- **$q_{\text{acc}}$**

Input $w$

... $a$ $a$ $a$ $a$ ...

Universal TM $U$

- **start**
  - **$q_0$**
  - **$q_1$**
  - **$q_{\text{rej}}$**

Input $\langle M, w \rangle$

... $q_0$ $a$ $\square$ $R$ ... $q_1$ $a$ ... $a$ $a$ $a$ $a$ ...

- Look at next char of $w$
- If $M$ is in accepting state
- If $M$ is in rejecting state
- Look up what $M$ should do upon reading $w$
- Update state and tape
Machine $M$

Input $w$

Universal TM $U$

Input $\langle M, w \rangle$

- If $M$ is in accepting state:
  - Look at next char of $w$
  - Update state and tape

- If $M$ is in rejecting state:
  - Look up what $M$ should do upon reading $w$
Machine $M$

Input $w$

Universal TM $U$

Input $\langle M, w \rangle$

- **Machine $M$**
  - Start state: $q_0$
  - States: $q_0, q_1, q_{\text{rej}}$
  - Transitions:
    - $q_0 \xrightarrow{\text{a}} q_1$
    - $q_1 \xrightarrow{\text{a}} q_{\text{rej}}$
    - $q_{\text{rej}} \xrightarrow{\Box} q_{\text{rej}}$
  - Accepting state: $q_{\text{acc}}$
  - Rejection state: $q_{\text{rej}}$

- **Universal TM $U$**
  - Start state: $q_{\text{acc}}$
  - States: $q_{\text{acc}}, q_{\text{rej}}$
  - Transitions:
    - If $M$ is in accepting state:
      - Look at next char of $w$
    - If $M$ is in rejecting state:
      - Update state and tape
  - Actions:
    - Look up what $M$ should do upon reading $w$

- **Input $w$**
  - Input: $\cdots \text{a a a} \cdots$

- **Input $\langle M, w \rangle$**
  - Universal TM:
    - States: $q_0, q_1, \Box, R$
    - Tape: $\cdots \text{a a a} \cdots$
  - Machine $M$:
    - States: $\cdots q_0 a \Box R \cdots q_1 a \cdots$
    - Input $w$: $\cdots \text{a a a} \cdots$
Machine $M$

Universal TM $U$

Input $w$

Input $\langle M, w \rangle$

If $M$ is in accepting state

If $M$ is in rejecting state

Look at next char of $w$

Look up what $M$ should do upon reading $w$

Update state and tape
**Machine $M$**

State transition diagram for $M$:
- Start state $q_0$ with transition $\square \rightarrow \square, R$.
- State $q_1$ with transition $a \rightarrow \square, R$.
- Transition $a \rightarrow \square, R$ from $q_0$.
- Transition $\square \rightarrow \square, R$ from $q_1$.

**Universal TM $U$**

State transition diagram for $U$:
- Start state $q_0$.
- Transition $a \rightarrow \square, R$.
- Transition $\square \rightarrow \square, R$.
- If $M$ is in accepting state, update state and tape.
- If $M$ is in rejecting state, update state and tape.
- Look up what $M$ should do upon reading $w$.
- Look at next char of $w$.

**Input $w$**

Input sequence: $\ldots a a a \ldots$

**Input $\langle M, w \rangle$**

Input sequence: $\ldots q_0 a \square R \ldots q_1 a \ldots a a a \ldots$

**M**

$M$

**w**

$w$
As a high-level description:

$$U = \text{"On input } \langle M, w \rangle, \text{ where } M \text{ is a TM and } w \in \Sigma^*, \text{ Run } M \text{ on } w. \text{ If } M \text{ accepts } w, U \text{ accepts } \langle M, w \rangle. \text{ If } M \text{ rejects } w, U \text{ rejects } \langle M, w \rangle."}$$

If $M$ loops on $w$, then $U$ loops as well. This is implicit in the description.
The language of $U$

$U$, 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 $U$ as a recognizer, it does recognize some language.

What language is that?
The language of $U$

$U$, 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 $w$.

$L(U) = \{ \langle M, w \rangle | M$ is a TM and $M$ accepts $w \}$

$= \{ \langle M, w \rangle | M$ is a TM and $w \in L(M) \}$
The acceptance language for Turing machines, denoted $A_{TM}$, is the language of the universal Turing machine:

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

Because there is a Turing machine, $U$, that recognizes $A_{TM}$, we know $A_{TM} \in \text{RE}$. 
All languages

CFLs

Regular languages

RE

ATM

All languages
Why do we care about universality?
Reason 1: It has practical consequences
What happens if we replace the Turing machine input with a normal computer program?
What happens if we replace the Turing machine input with a normal computer program?
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* (or *emulator*) is a program that simulates an entire operating system.

- Virtual machines are used in computer security, cloud computing, and even by individual end users.
Party like it’s 1999 1990

What are the first names of the four other members in your party?

1. Alan
2. Turing
3. Alonzo
4. Church
5. Post

(Enter names or press Enter)

archive.org/details/msdos_Oregon_Trail_The_1990
It’s not a coincidence that interpreters and virtual machines are possible – Turing’s 1936 paper says that any general-purpose computing system must be able to do this!
The key idea behind the universal TM is that TMs can be fed as inputs to other TMs.

Similarly,

an interpreter is a program that takes other programs as inputs, and
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 15 May 1951, Alan Turing delivered a radio lecture on the BBC, where he argued that “it is not altogether unreasonable to describe digital computers as brains”.

Why would he think this, given the very limited abilities of computers of the time?
“I should also say that ‘If any machine can be appropriately described as a brain, then any [i.e., every] 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. If it is accepted that real brains, as found in animals, and in particular in men, are a sort of machine it will follow that our digital computer suitably programmed, will behave like a brain.”
“This argument involves several assumptions which can quite reasonably be challenged.”

Alan Turing, 1951
Self-referentiality
We can write programs that act on themselves.

As a fun (but tricky) case, we can consider *quines* — programs that, when run, print out their own source code.

How would you write such a program?
selfie.py:

```python
my_source = open("selfie.py").read()
print(my_source)
```
It’s cheating to read the source file – and what happens if someone changes the file after the program starts?
Coding break!
to_print = [
    ...
]

for line in to_print:
    print(line)
We need to substitute the entire source code here, but without getting sucked into infinite recursion.
to_print = [
"to_print = [",
  "@",
  "\n",
  "for line in to_print:",
  "  print(line)"]
]

for line in to_print:
  if line == "@":
    ...
  else:
    print(line)
to_print = [
    "to_print = [",
    "@",
    "]",
    "for line in to_print;",
    "    print(line)",
]

def print_program_in_quotes():
    ...

for line in to_print:
    if line == "@":
        print_program_in_quotes()
    else:
        print(line)
to_print = [
    "to_print = [",
    "@",
    "]",
    "",
    "for line in to_print:",
    "  print(line)",
]

def print_program_in_quotes():
    for line in to_print:
        print("  " + line + ",")

for line in to_print:
    if line == "@":
        print_program_in_quotes()
    else:
        print(line)
Close! But we need to quote the strings in `to_print` and escape quotation marks inside them!
```python
import json

to_print = [
    "import json",
    "",
    "to_print = [",
    ""
]

"def print_program_in_quotes():",
  "for line in to_print:",
  "  print(" + json.dumps(line) + ",")",
=
"for line in to_print:",
  "if line == ":",
  "  print_program_in_quotes()",
  "else:",
  "  print(line"
]

def print_program_in_quotes():
  for line in to_print:
    print(" + json.dumps(line) + ",")

for line in to_print:
  if line == "":
    print_program_in_quotes()
  else:
    print(line)
```
A self-referential program doesn’t need to print its source code; we can make it a string that we look at in order to compute!

```python
import json

to_print = [
    "import json",
    ",",
    "to_print = [",
    "",
    "],",
    ",",
    "def my_source():",
    "    result = "\n",
    "    for line in to_print:",
    "        if line == "\@":",
    "            for line in to_print:
                result += json.dumps(line) + "\n",
        else:
            result += line + "\n",
    "    return result",
    ",",
    "if len(my_source()) % 2 == 0:",
    "    print("I am an even-length program."),",
    "else:",
    "    print("I am an odd-length program."),",
]

def my_source():
    result = ""
    for line in to_print:
        if line == "@":
            for line in to_print:
                result += json.dumps(line) + "\n"
        else:
            result += line + "\n"
    return result

if len(my_source()) % 2 == 0:
    print("I am an even-length program.")
else:
    print("I am an odd-length program.")
```

cs.vassar.edu/~cs240/class/obtain_source.py
The fact that we can write quines isn’t a coincidence!

**THEOREM** It is possible to construct TMs that perform arbitrary computations on their own “source code”.

In other words, any computing system that’s equal to a Turing machine possess some mechanism for self-reference.
CLAIM  Going forward, assume that any program can be augmented to include a `my_source()` function that returns a string representation of its source code.

This means we can write programs like these:

```python
def narcissist(input: str) -> bool:
    me = my_source()
    return input == me

def accept_longer_strings(input: str) -> bool:
    me = my_source()
    return len(input) > len(me)
```
Teaser: Self-reference lets machines compute on themselves. That will let them do cruel and unusual things.
PINOCCHIO

MY NAME ISN'T PINOCCHIO.

PRANSON REESE

WOW!
We now stand at the foot of the mountains of undecidability.

Next time we’ll see how the emergent properties of universality and self-reference bring us to the limits of computation!
Acknowledgments

This lecture incorporates material from:

Amy Liu, Stanford University
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*