Self-knowledge and a program that prints itself
Can we ever really know ourselves?

- Your brain has 10 billion neurons.
- Is it possible to know/remember/represent all those neural connections in that same brain?
- On average, you have less than 1 neuron to use to store the info about each neuron.

Can a program ever know/represent/process on itself?

Yes!
A PARADOX IN THE EARLY DAYS OF BIOLOGY:

FROM ONE ANIMAL, IS BORN ANOTHER "COPY."

EACH PARENT CONTAINS SOME TINY, TINY LITTLE PEOPLE, WHO MERELY GROW INTO FULL INDIVIDUALS.

BUT WHAT ABOUT THEIR CHILDREN?

.... NAH!
Computer Viruses.

Must make a copy of themselves.
For simplicity, just imagine printing a copy of yourself.

Approach #1

Use a pointer, P.
Make P point to first byte of the code
For 157 times...
Print *P
P := P + 1
End

Countermeasure:
The O.S. flags memory as executable
* Read/Write.
The machine prevents all programs from "reading" code bytes.

Solution used by biological life:
DNA: "code" which is "executed" to produce proteins
DNA: Used as "data" when DNA is copied.

Approach #2

Recursion theorem
Computer Viruses.
GOAL: WRITE A PROGRAM THAT PRINTS ITSELF.

OUR PROGRAMMING LANGUAGE?
 VARIABLES, ASSIGNMENT
 STRINGS
 PRINT STATEMENTS.
   x ← 'world'
 PRINT 'Hello'
 PRINT x

WE'LL IGNORE
 * PRINTING NEWLINES
 * ESCAPING QUOTES.
   ' 4 o' clock'
   ' 4 o\' clock'
 9 H 4 o' clock 9 chars

PROGRAMS THAT PRINT THEMSELVES ARE CALLED "QUINES".

TURING MACHINES:
The tape is used as MEMORY.

MODERN COMPUTERS:
Variables are used for MEMORY.
**STEP 1:**

\[ x \rightarrow ? \]

PRINT \( x \)

**STEP 2:**

\[ x \rightarrow ? \]

PRINT \( 'x 4' \)

PRINT \( x \)

PRINT \( \ldots \ldots \ldots \)
**STEP 4:**

\[ x \leftarrow \text{"PRINT 'X=' PRINT X PRINT "" PRINT X""}ight. \\
\text{PRINT 'X='}
\text{PRINT X}
\text{PRINT \"\"}
\text{PRINT X}

**EXECUTING THIS:**

\[ x \leftarrow \text{"PRINT 'X=' PRINT X PRINT "" PRINT X""}ight. \\
\text{PRINT 'X='}
\text{PRINT X}
\text{PRINT \"\"}
\text{PRINT X}
In the "C" Language

Note: ASCII 34 = double quote char("")

```c
printf("Hello %c %s %c", 34, "World", 34);
Hello "World"
```

```c
main() {
    char *x = "";
    printf(x, 34, x, 34);
    printf(x, 34, x, 34);
}
```
QUINE:
A T.M. THAT PRINTS ITS OWN DESCRIPTION
APPROACH TO IMPLEMENTING "quine" ON A T.M.

Break the task into 2 steps.

**step a:**

\[ X \leftarrow \langle \text{step b} \rangle \]

**step b:**

print out \langle \text{step a} \rangle  
(we get to use \( X \)!)  

print out \langle \text{step b} \rangle  
(we get to use \( X \)!)  

**goals:**

Each step is a T.M./subroutine.  
Execute step a, then step b.  
This should write out

\[
\langle \text{step a} \rangle \quad | \quad \langle \text{step b} \rangle \quad | \quad \ldots
\]

Our "quine" TM.
Let $P_w$ be a Turing Machine that prints out $w$.

$P_{10110}$ writes $10110$ on the tape.

Let $\langle P_w \rangle$ be the representation of Turing Machine $P_w$.

Given a string $w$, could you build a T.M. to write out $w$?

SURE! EASY!

$P_w$ looks like $\xrightarrow{0} \xrightarrow{0} \xrightarrow{\cdots} \xrightarrow{\text{Acc}}$

This task is clearly computable. A computable function $g$ does it

$g: \Sigma^* \to \Sigma^*$

$g(w) \to \langle P_w \rangle$

Given a string $w$ on the tape.
**STEP A:**

Write a long string on the tape. Call this string $X$.

The string will turn out to be $\langle$STEP B$\rangle$.

We don't know the string yet. We can't finish coding STEP A yet.

Once we know $X$, we can easily finish coding step A.

**IN FACT, WE CAN JUST USE $g(x)$ TO DO THE ENTIRE CODING OF STEP A, ONCE WE KNOW $X$!!**
**STEP B:**

When it starts, the tape contains a long string \( \times \).

**MAKE A COPY OF** \( \times \):

\[
\begin{array}{ccc}
\times & X & 1 \\
\end{array}
\]

Use \( g \) as a subroutine.

Call it on \( X \) to compute \( g(X) \)

Note: \( g(X) = \langle \text{STEP A} \rangle \)

\[
\begin{array}{ccc}
\langle \text{STEP A} \rangle & X & \ldots
\end{array}
\]

And Now the Magic!

Let \( X \) be the description of **STEP B**. \( X = \langle \text{STEP B} \rangle \)

Our tape contains:

\[
\begin{array}{ccc}
\langle \text{STEP A} \rangle & \langle \text{STEP B} \rangle & \ldots
\end{array}
\]

We are done coding **STEP B**.
We now know \( \langle \text{STEP B} \rangle \).
Go back and finish up coding **STEP A**.
THE
RECURSION
THEOREM
OPERATIONS YOU MIGHT DO ON A T.M.

* COUNT THE NUMBER OF STATES.
* CHECK TO SEE IF ACCEPT IS EVEN REACHABLE FROM INITIAL STATE.
* CHECK TO SEE IF THE T.M. ACCEPTS $w$.
* etc.

**APPROACH:**

Build a TM to do this

$$t(\langle M \rangle, w)$$

Perhaps you'd like to run this TM on itself:

$$t(\langle t \rangle, w)$$

You have to pass $t$ its own description.

Is there any other way?
YES
You can build a T.M. \( r \) that does exactly what \( t \) would do if passed a description of itself.
\[ r(w) = t(<r>, w) \]

RECURSION THEOREM
If you can build \( t \), then there exists another T.M. that does the same thing but computes its own DESCRIPTION instead of having to take it as an input.
Recursion Theorem

Let $T$ be some Turing machine that computes some function $t$.

$t : \Sigma^* \times \Sigma^* \rightarrow \Sigma^* \quad / \quad t(\langle M \rangle, w)$

Then there will always exist another Turing Machine $R$ that does the same thing as $t$ when $t$ is applied to a description of itself.

That is $R$ computes the function $r : \Sigma^* \rightarrow \Sigma^*$

and for every $w$...

$r(w) = t(\langle R \rangle, w)$
The Recursion Theorem: Some Results
**Bottom Line**

Whenever we are specifying a Turing Machine algorithm, we can say:

Obtain, via the recursion theorem, a description of self, \( \langle \text{SELF} \rangle \).

Or, more concisely,

\[ X \leftarrow \langle \text{SELF} \rangle \]

---

**The Quine Program (Short Version):**

\[
\begin{array}{l}
X \leftarrow \langle \text{SELF} \rangle \\
\text{PRINT } X \\
\end{array}
\]

The recursion theorem says: this is an entirely legal T.M. program.
Theorem (previously proven)

\[ A_{TM} = \exists \langle M, w \rangle \mid M \text{ is a } \overline{T} \text{, } M \text{ accepts } w \exists \]

is undecidable

New proof

Assume \( H \) decides \( A_{TM} \).

Construct machine \( B \):

**Input:** \( w \)

\[ x \rightarrow \langle B \rangle \]

Run \( H \) on \( \langle B \rangle, w \)

Do the opposite:

- if \( H \) accepts \( \) THEN REJECT
- if \( H \) rejects \( \) THEN ACCEPT

Running \( B \) on input \( w \) does the opposite of what \( H \) says \( B \) does. Therefore \( H \) is wrong. \( H \) can't be deciding \( A_{TM} \).
THE "SIZE" OF A TURING MACHINE:

$|\langle M \rangle| = \text{Number of symbols in the description of } M.$

**DEFINITION**

A Turing Machine $M$ is "MINIMAL" if there is no Turing machine equivalent to $M$ with a shorter description.

What about the set of MINIMAL Turing Machines?
The set

$$\text{MIN}_{\text{Tm}} = \{ \langle M \rangle \mid M \text{ is a "minimal" Turing Machine} \}$$

is NOT TURING-RECOGNIZABLE.
PROOF

Assume \( M^{TM} \) is Turing Recognizable.
Then \( \exists \) an enumerator \( E \)
that will list them out.
Use \( E \) to construct a new
machine "C", as follows:

\[ C \]

**Input:** \( w \)

**Algorithm:**
- Obtain, via Recursion Theorem, a
description of self, \( \langle C \rangle \).
- Run \( E \) until it prints
out a machine \( D \) with
a longer description than \( C \).
- Simulate \( D \) on \( w \).

\( M^{TM} \) is infinite, so we'll eventually
find a machine \( D \) which is longer.
\( C \) simulates \( D \); therefore they are equivalent.
\( D \) cannot be minimal
CONTRACTION!
Fixed Points
And
Turing Machines
DEFINITION

A "fixed point" of a function is a value that is unchanged by repeated applications of the function.

Example

\[ \begin{array}{c|c}
 x & f(x) \\
 1 & 5 \\
 2 & 4 \\
 3 & 6 \\
 4 & 4 \\
 5 & 2 \\
 6 & 4 \\
\end{array} \]

Example

SPACE OF VALUES

FUNCTION

---
FUNCTIONS: Computable Transformations.

VALUES: Turing Machine Descriptions.

**ANOTHER VERSION OF THE RECURSION THEOREM (The "Fixed Point" Version)**

For any transformation function on Turing Machines,

There will always exist a Turing Machine which is unchanged by the Transformation.
Theorem

Let $t$ be any computable function

$t: \Sigma^* \rightarrow \Sigma^*$

[We can apply $t$ to descriptions of]
Turing Machines: $t(<M>)$

Then there is a Turing Machine $F$ such that

$t(<F>)$ is equivalent to $F$.

Proof

Let $F$ be the following TM:

**Input:** $w$

**Algorithm:**

- Obtain description of self, $<F>$.
- Compute $t(<F>)$. To obtain a new Turing Machine, $G$
- Simulate $G$ on $w$.

$G$ and $F$ are equivalent.

$<G> = t(<F>)$

So $<F>$ and $t(<F>)$ are equivalent.