# Unsolvable Problems

Part Two

# Recap from Last Time

# What does this program do?

```
bool willAccept(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, input)) {
      reject();
   } else {
      accept();
   }
}
```

What happens if…

… this program accepts its input?

It rejects the input!

… this program doesn't accept its input?

It accepts the input!

# What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

What happens if…

… this program halts on this input?
It loops on the input!

… this program loops on this input?
It halts on the input!

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   bool actualAnswer =
       countRs(input) > countDs(input);

   if (isSecureVotingMachine(me)) {
      return !actualAnswer;
   } else {
      return actualAnswer;
   }
}
```

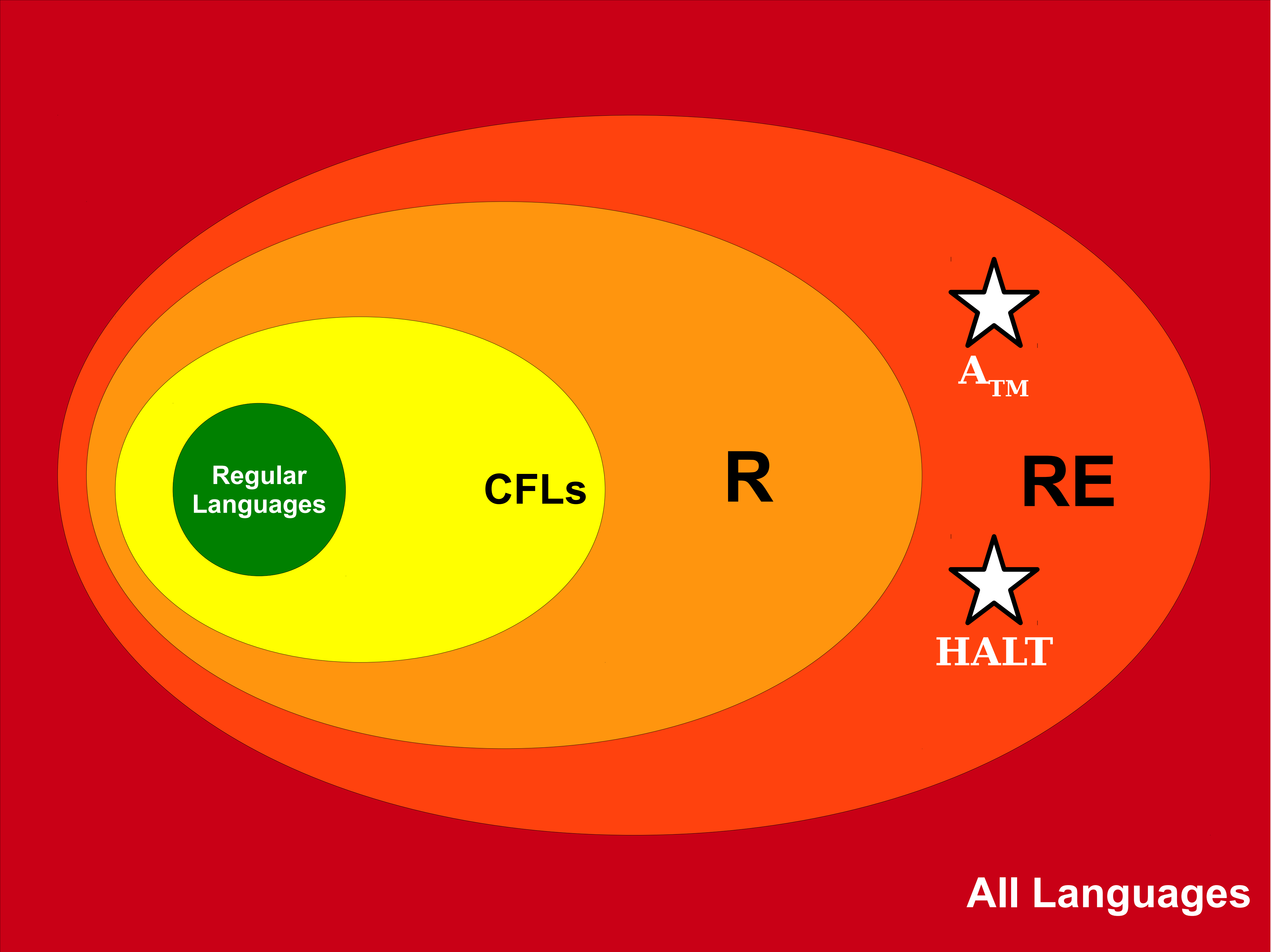
What happens if...

... this program is a secure voting machine?

It's not a secure machine!

... this program is not a secure voting machine?

It is a secure voting machine!


Regular Languages
CFLs
R
RE
$A_{TM}$
HALT
All Languages

# Verifiers

- A ***verifier*** for a language $L$ is a TM $V$ with the following properties:
  - $V$ is a decider (that is, $V$ halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

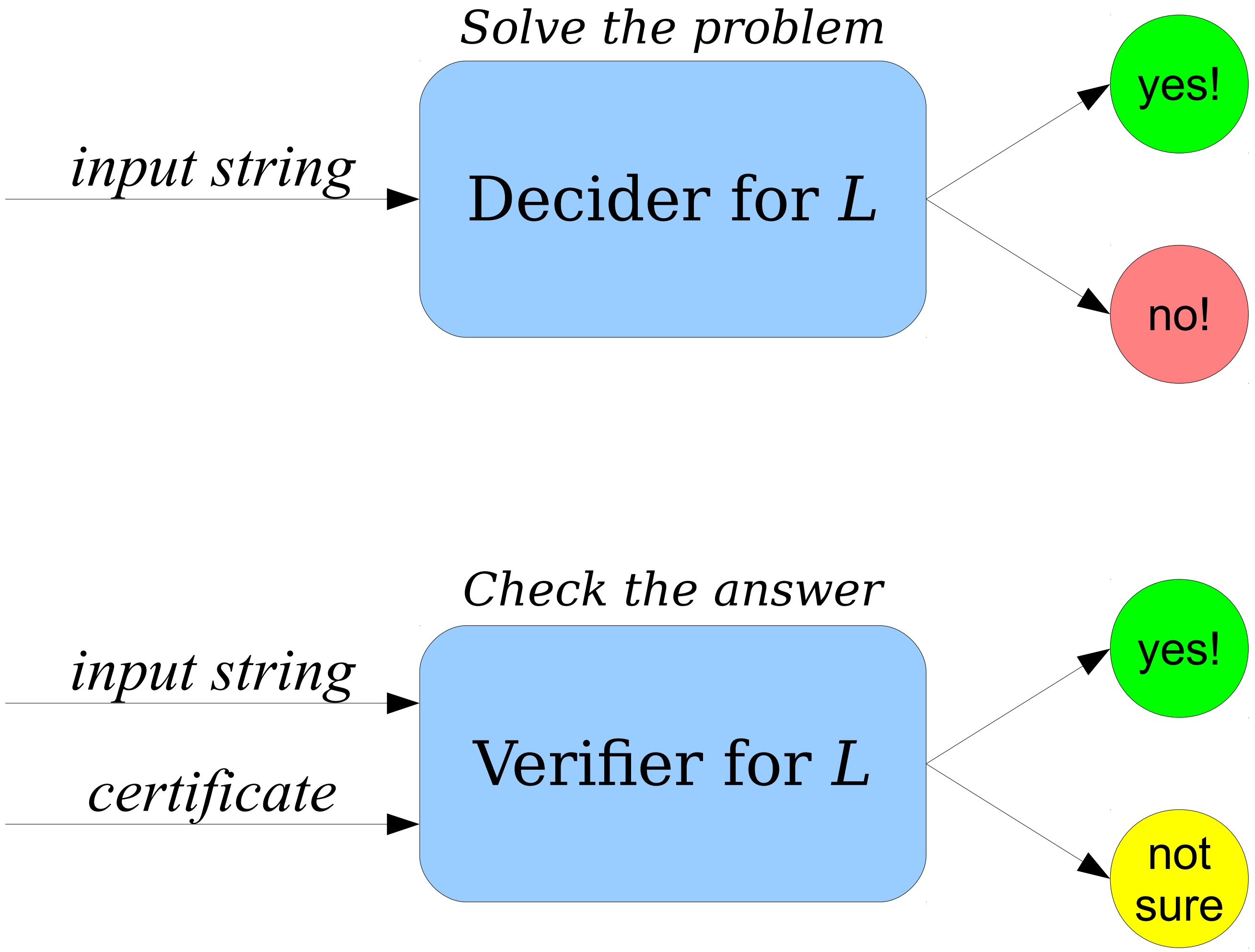
$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \textbf{ accepts } \langle w, c\rangle}$$

- Some notes about $V$:
  - If $V$ accepts $\langle w, c\rangle$, then we're guaranteed $w \in L$.
  - If $V$ does not accept $\langle w, c\rangle$, then either
    - $w \in L$, but you gave the wrong $c$, or
    - $w \notin L$, so no possible $c$ will work.

# New Stuff!

# So... What's a Verifier?

# Deciders and Verifiers



*Solve the problem*

*input string*

Decider for $L$

yes!

no!

*Check the answer*

*input string*

*certificate*

Verifier for $L$

yes!

not sure

# Verification

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 6 | | 1 | | |
| | | | | | 3 | | 5 | 2 |
| 3 | | | 1 | | 5 | 9 | | 7 |
| 6 | | 5 | | 3 | | 8 | | 9 |
| | 1 | | | | | | 2 | |
| 8 | | 2 | | 1 | | 5 | | 4 |
| 1 | | 3 | 2 | | 7 | | | 8 |
| 5 | 7 | | 4 | | | | | |
| | | 4 | | 8 | | 7 | | |

Does this Sudoku puzzle
have a solution?

# Verification

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 7 | 1 | 6 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 3 | 1 | 5 | 2 |
| 3 | 1 | 1 | 1 | 1 | 5 | 9 | 1 | 7 |
| 6 | 1 | 5 | 1 | 3 | 1 | 8 | 1 | 9 |
| 1 | 1 | 1 | 1 | 4 | 1 | 1 | 2 | 1 |
| 8 | 1 | 2 | 1 | 1 | 1 | 5 | 1 | 4 |
| 1 | 1 | 3 | 2 | 1 | 7 | 1 | 1 | 8 |
| 5 | 7 | 1 | 4 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 4 | 1 | 8 | 1 | 7 | 1 | 1 |

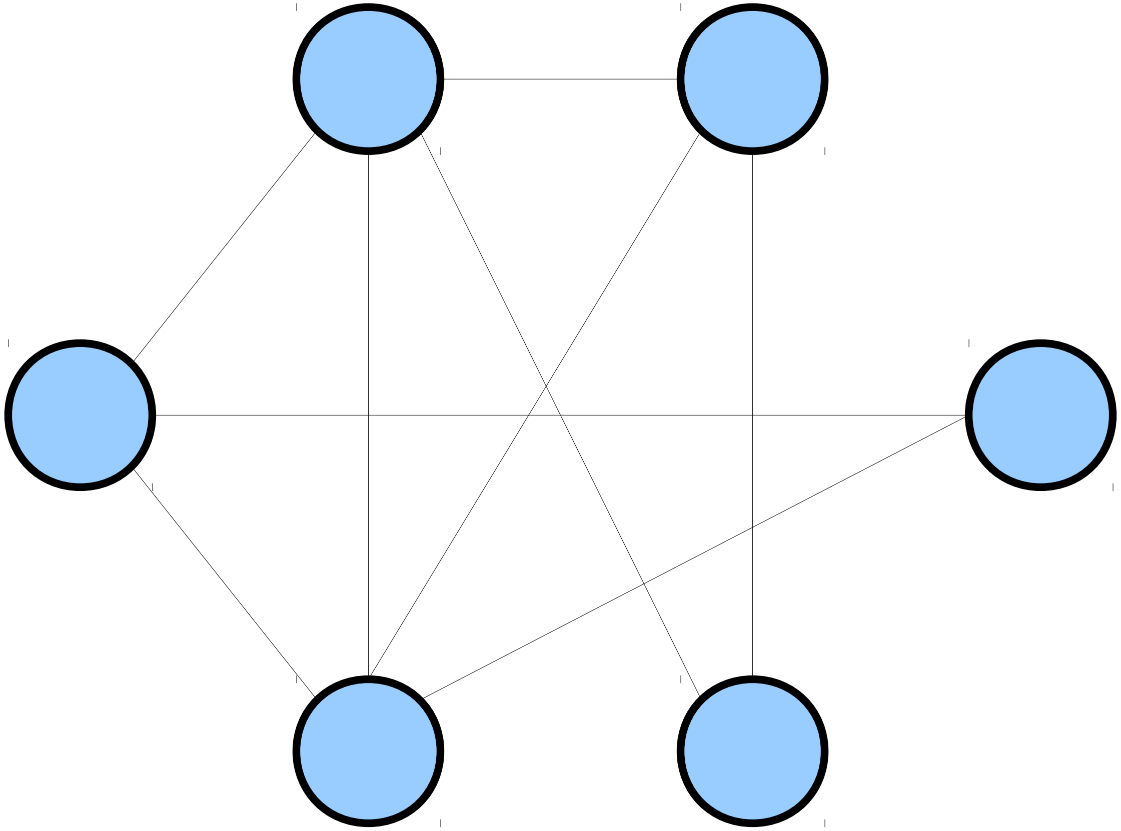
Does this Sudoku puzzle
have a solution?

# Verification

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 5 | 7 | 9 | 6 | 4 | 1 | 8 | 3 |
| 4 | 9 | 1 | 8 | 7 | 3 | 6 | 5 | 2 |
| 3 | 8 | 6 | 1 | 2 | 5 | 9 | 4 | 7 |
| 6 | 4 | 5 | 7 | 3 | 2 | 8 | 1 | 9 |
| 7 | 1 | 9 | 5 | 4 | 8 | 3 | 2 | 6 |
| 8 | 3 | 2 | 6 | 1 | 9 | 5 | 7 | 4 |
| 1 | 6 | 3 | 2 | 5 | 7 | 4 | 9 | 8 |
| 5 | 7 | 8 | 4 | 9 | 6 | 2 | 3 | 1 |
| 9 | 2 | 4 | 3 | 8 | 1 | 7 | 6 | 5 |

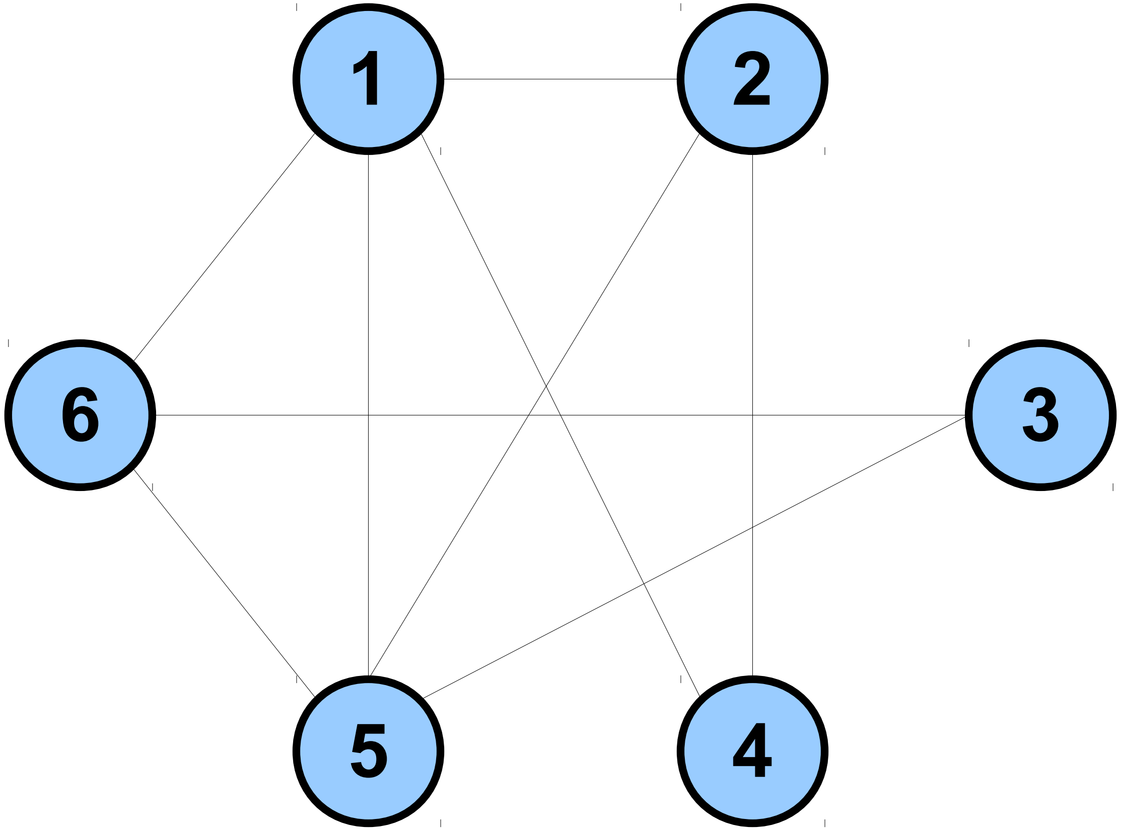
Does this Sudoku puzzle
have a solution?

# Verification



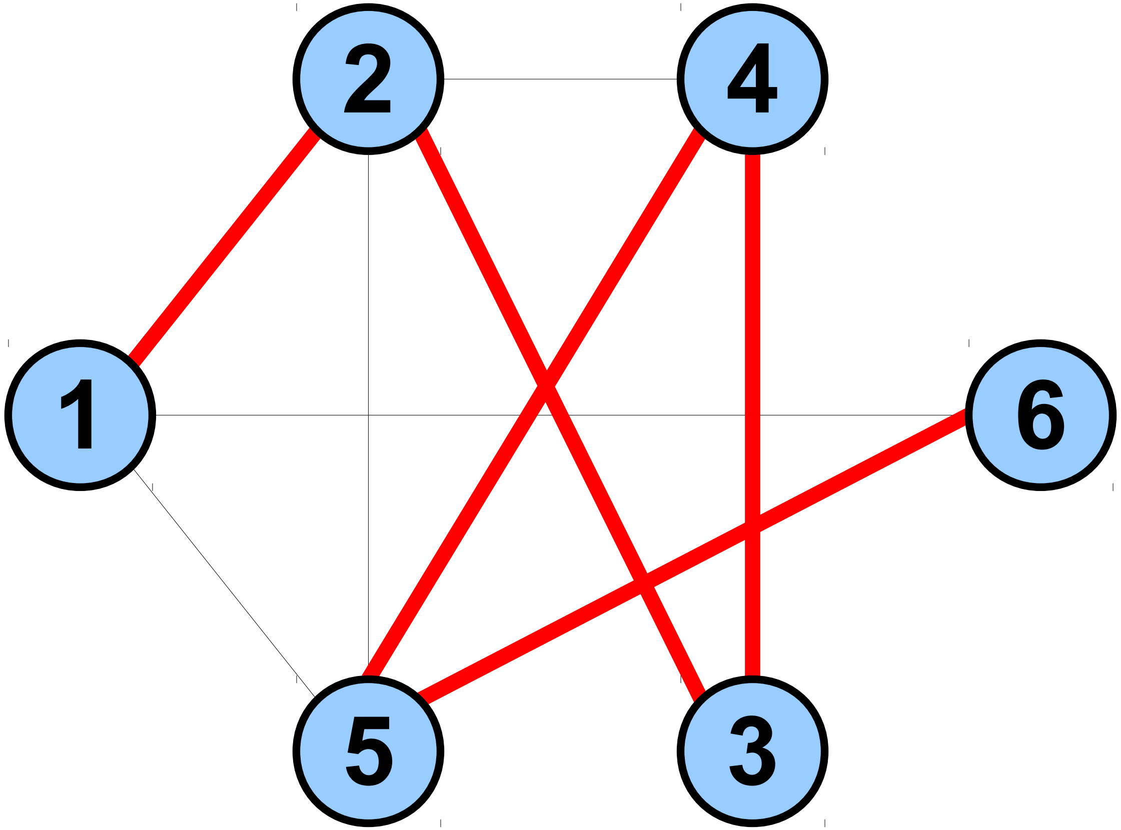
Is there a simple path that goes through every node exactly once?

# Verification



Is there a simple path that goes through every node exactly once?

# Verification



Is there a simple path that goes through every node exactly once?

# Verification

**11**

Does the hailstone sequence
terminate for this number?

# Verification

**11**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**34**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**17**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**52**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**26**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**13**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**11**

Does the hailstone sequence
terminate for this number?

# Verification

**11**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**34**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**17**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**52**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**26**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**13**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**40**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**20**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**10**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**5**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**16**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**8**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**4**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**2**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**1**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# What's In Common?

- These verification procedures *do not* try to solve the problem. They just check a candidate answer.
- These verification procedures will always terminate in finite time. We'll never have to wait forever.
- If the candidate answer is right, we're *convinced* that the answer is “yes.” If the candidate answer is wrong, we don't learn much.

# What languages are verifiable?

***Theorem:*** If $L$ is a language, then there is a verifier for $L$ if and only if $L \in$ **RE**.

# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.
- Imagine you have a string $w$ and you have the verifier $V$. You know that if $w \in L$, then there is some $c \in \Sigma^*$ where $V$ will accept $\langle w, c \rangle$. You also know that if $w \notin L$, then there is *no* choice of $c$ such that $V$ accepts $\langle w, c \rangle$.
- You need to build a recognizer $M$ such that if $w \in L$, then $M$ accepts $w$, and if $w \notin L$, then $M$ does not accept $w$.
- We know nothing about $L$ except that $V$ exists and how $V$ behaves. Therefore, to see whether $w \in L$, we need to see whether any certificate $c$ exists that causes $V$ to accept $\langle w, c \rangle$.
- ***Idea:*** What if we try running $V$ on every possible certificate?

# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof idea:*** Build a recognizer that tries every possible certificate to see if $w \in L$.
- ***Proof sketch:*** Show that this TM is a recognizer for $L$:

```
$M$ = “On input $w$:
          For $i = 0$ to $\infty$:
              For each string $c$ of length $i$:
                  Run $V$ on $\langle w, c \rangle$.
                  If $V$ accepts $\langle w, c \rangle$, $M$ accepts $w$.”
```

# Verifiers and **RE**

- ***Theorem:*** If $L \in$ **RE**, then there is a verifier for $L$.
- ***Proof goal:*** Beginning with a recognizer $M$ for the language $L$, show how to construct a verifier $V$ for $L$.
- The machine $M$ will accept $w$ if $w \in L$. If $M$ doesn't accept $w$, then $w \notin L$.
- A certificate is supposed to be some sort of “proof” that the string is in the language. Since the only thing we know about $L$ is that $M$ is a recognizer for it, our certificate would have to tell us something about what $M$ does.
- We need to choose a certificate with the following properties:
  - We can decide in finite time whether a certificate is right or wrong.
  - A “good” certificate proves that $w \in L$ (meaning $M$ accepts $w$)
  - A “bad” certificate never proves $w \in L$.
- ***Idea:*** If $M$ accepts $w$, it will do so in finitely many steps. What if our certificate is the number of steps?

# Verifiers and **RE**

- ***Theorem:*** If $L \in$ **RE**, then there is a verifier for $L$.
- ***Proof sketch:*** Let $L$ be an **RE** language and let $M$ be a recognizer for it. Then show that this is a verifier for $L$:

```
V = “On input ⟨$w$, $n$⟩, where $n \in \mathbb{N}$:
            Run $M$ on $w$ for $n$ steps.
            If $M$ accepts $w$ within $n$ steps, accept.
            If $M$ did not accept $w$ in $n$ steps, reject.”
```

# **RE** and Proofs

- Verifiers and recognizers give two different perspectives on the “proof” intuition for **RE**.
- Verifiers are explicitly built to check proofs that strings are in the language.
  - If you know that some string *w* belongs to the language and you have the proof of it, you can convince someone else that $w \in L$.
- You can think of a recognizer as a device that “searches” for a proof that $w \in L$.
  - If it finds it, great!
  - If not, it might loop forever.

# **RE** and Proofs

- If the **RE** languages represent languages where membership can be proven, what does a non-**RE** language look like?
- Intuitively, a language is *not* in **RE** if there is no general way to prove that a given string $w \in L$ actually belongs to $L$.
- In other words, even if you knew that a string was in the language, you may never be able to convince anyone of it!

# Finding Non-**RE** Languages

# Finding Non-**RE** Languages

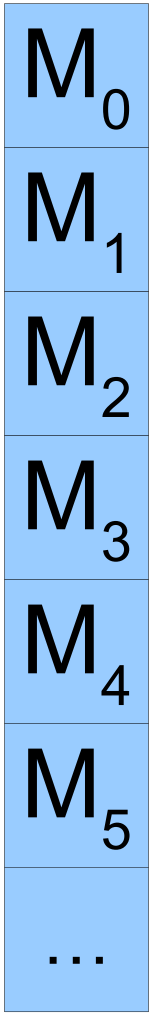
- Right now, we know that non-**RE** languages exist, but we have no idea what they look like.
- The first non-**RE** language we're going to see is one that is, essentially, constructed specifically to not be an **RE** language.
- Having seen that language, we'll then try to see if we can develop a more nuanced understanding of how to find non-**RE** languages.

# Languages, TMs, and TM Encodings

- Recall: The language of a TM $M$ is the set

$$\mathscr{L}(\boldsymbol{M}) = \{\ \boldsymbol{w} \in \boldsymbol{\Sigma}^* \mid \boldsymbol{M} \textbf{ accepts } \boldsymbol{w}\ \}$$

- Some of the strings in this set might be descriptions of TMs.
- What happens if we just focus on the set of strings that are legal TM descriptions?



$M_0$

$M_1$

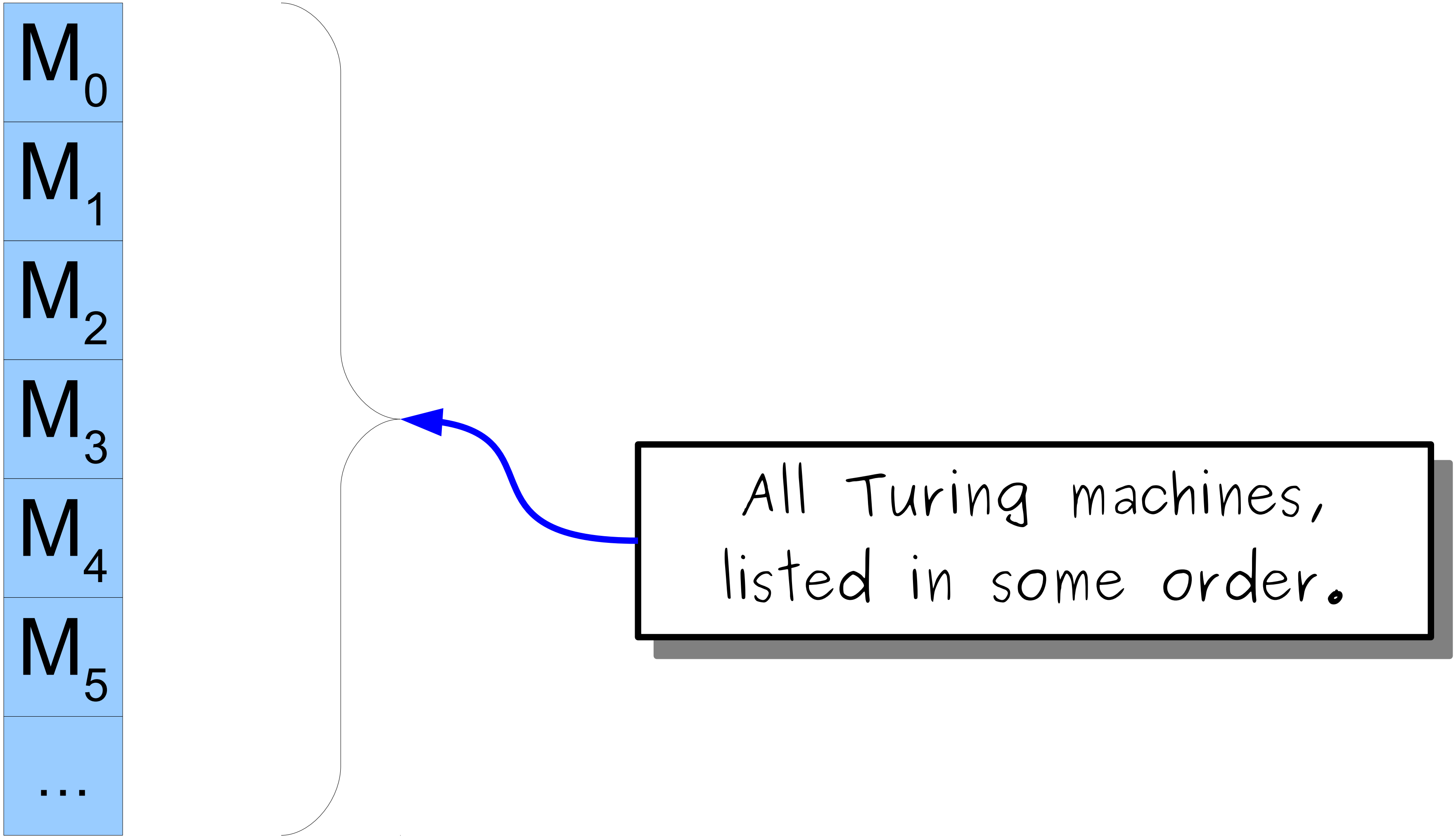
$M_2$

$M_3$

$M_4$

$M_5$

…



$M_0$

$M_1$

$M_2$

$M_3$

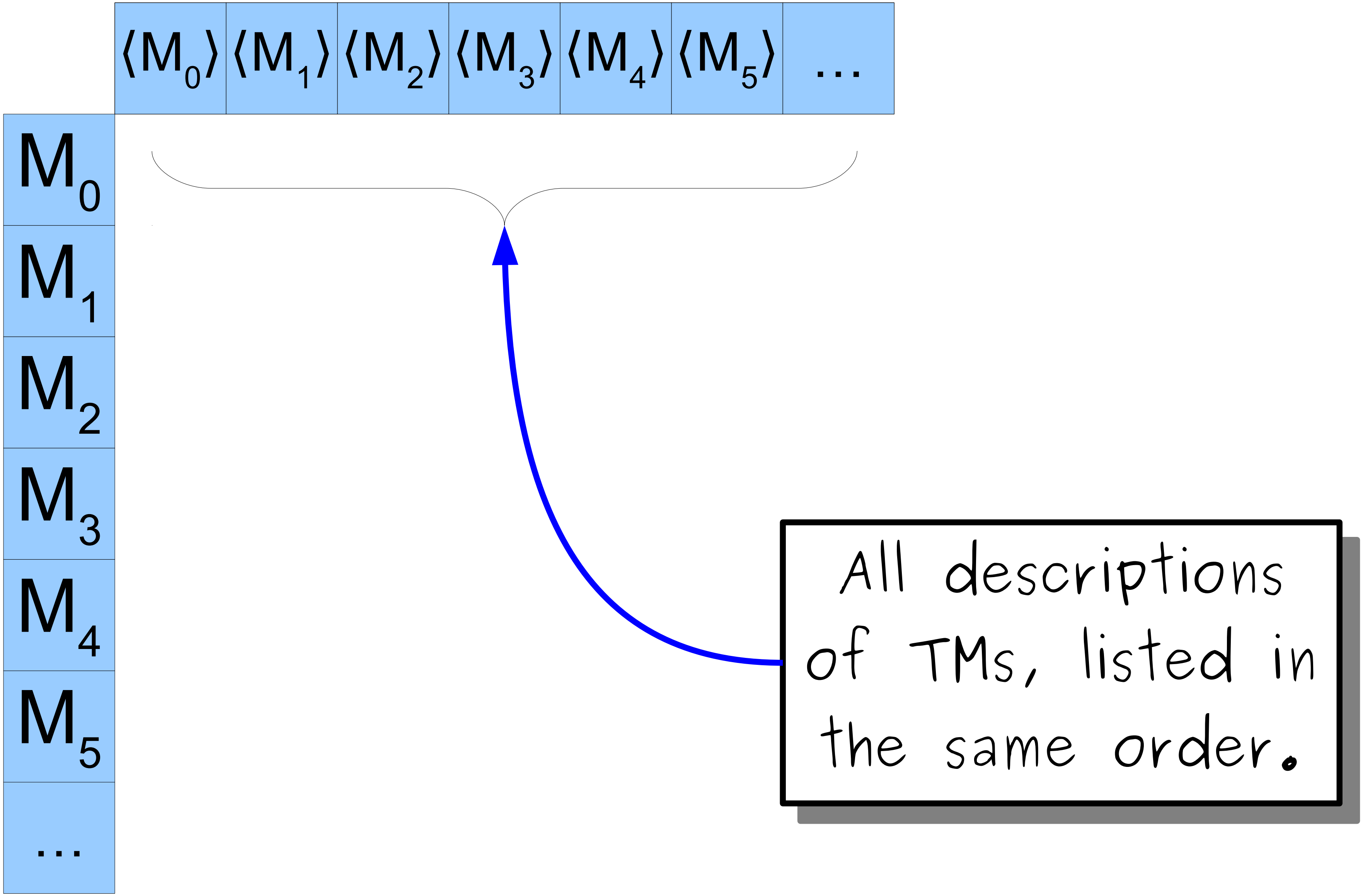
$M_4$

$M_5$

…

All Turing machines, listed in some order.

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ |  |  |  |  |  |  |  |
| $M_1$ |  |  |  |  |  |  |  |
| $M_2$ |  |  |  |  |  |  |  |
| $M_3$ |  |  |  |  |  |  |  |
| $M_4$ |  |  |  |  |  |  |  |
| $M_5$ |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |



All descriptions of TMs, listed in the same order.

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ |  |  |  |  |  |  |  |
| $M_2$ |  |  |  |  |  |  |  |
| $M_3$ |  |  |  |  |  |  |  |
| $M_4$ |  |  |  |  |  |  |  |
| $M_5$ |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ |  |  |  |  |  |  |  |
| $M_3$ |  |  |  |  |  |  |  |
| $M_4$ |  |  |  |  |  |  |  |
| $M_5$ |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ |  |  |  |  |  |  |  |
| $M_4$ |  |  |  |  |  |  |  |
| $M_5$ |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ |  |  |  |  |  |  |  |
| $M_5$ |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | | | | | | | |
| … | | | | | | | |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … |  |  |  |  |  |  |  |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| Acc | Acc | Acc | No | Acc | No | … |

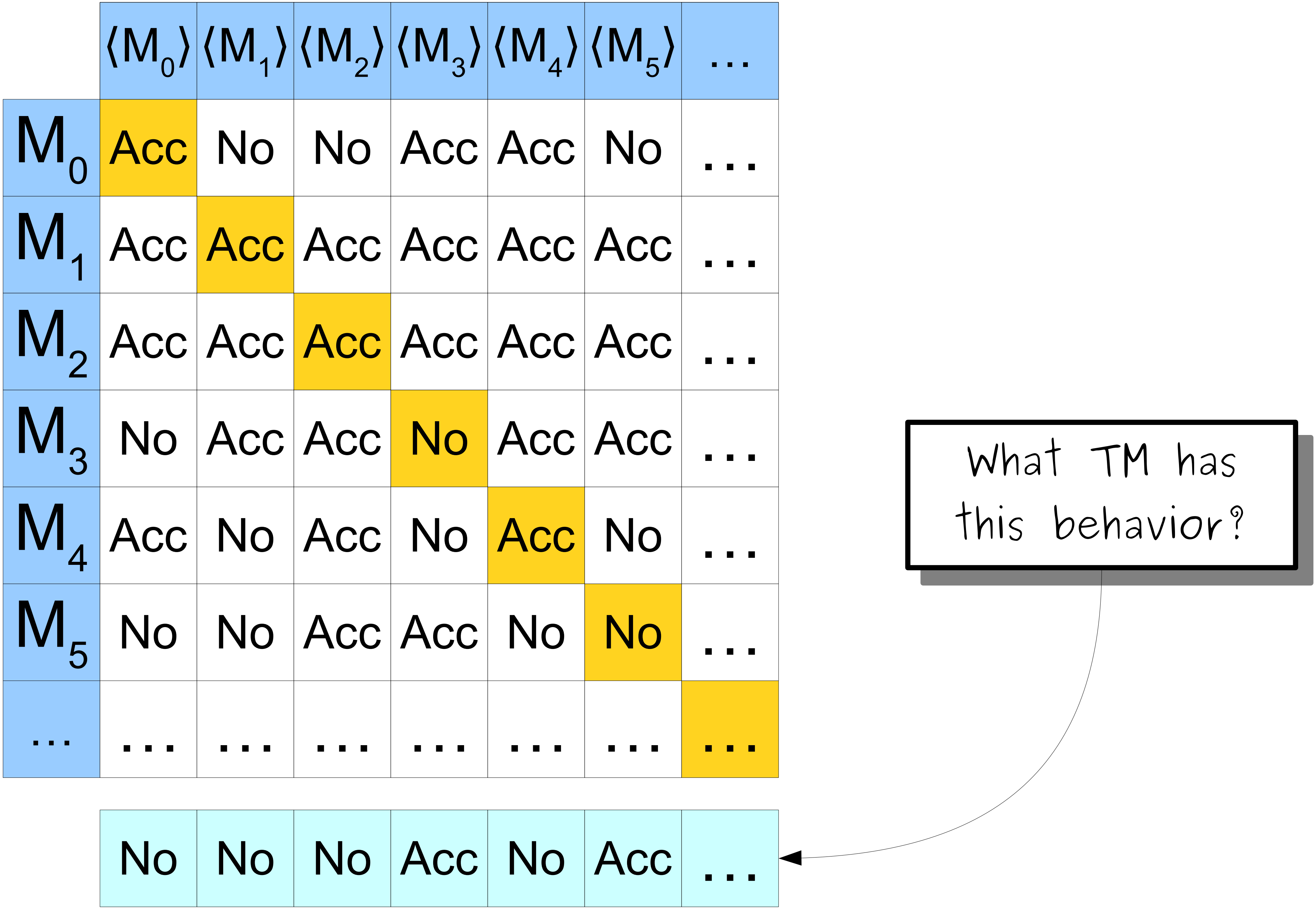
|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

Flip all "accept" to "no" and vice-versa

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |



| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

What TM has this behavior?

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

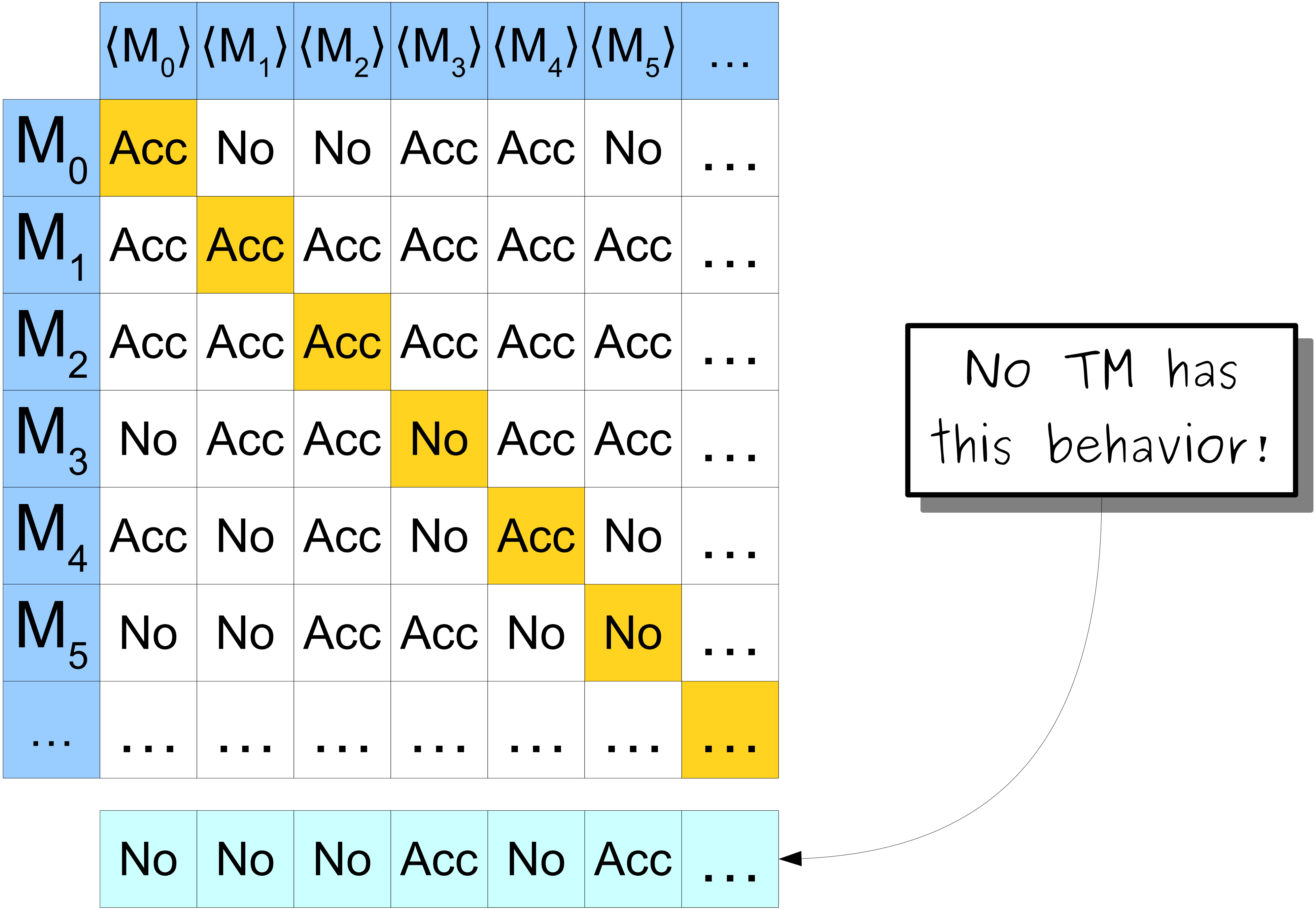
|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |



| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

No TM has this behavior!

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

**"The language of all TMs that do not accept their own description."**

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

**$\{ \langle M \rangle \mid M$ is a TM that does not accept $\langle M \rangle \}$**

|  | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

$$\{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$$

# Diagonalization Revisited

- The ***diagonalization language***, which we denote $\boldsymbol{L_\mathrm{D}}$, is defined as

$$\boldsymbol{L_\mathrm{D} = \{ \langle M \rangle \mid M \textbf{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}}$$

- That is, $L_\mathrm{D}$ is the set of descriptions of Turing machines that do not accept themselves.

$$\boldsymbol{L_{\mathrm{D}} = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}}$$

*Theorem:* $L_{\mathrm{D}} \notin$ **RE**.

# $L_D$ = { ⟨$M$⟩ | $M$ is a TM and ⟨$M$⟩ $\notin \mathscr{L}(M)$ }

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**.

## $L_D = \{ \langle M \rangle \mid M$ is a TM and $\langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_D$.

## $L_{\mathrm{D}} = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_{\mathrm{D}} \notin$ **RE**.

*Proof:* By contradiction; assume that $L_{\mathrm{D}} \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_{\mathrm{D}}$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_{\mathrm{D}} \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

# $L_D$ = { ⟨*M*⟩ | *M* is a TM and ⟨*M*⟩ ∉ $\mathscr{L}(M)$ }

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

Because $\mathscr{L}(R) = L_D$, we know that a string belongs to one set if and only if it belongs to the other.

# $\boldsymbol{L_{\mathrm{D}} = \{ \langle M \rangle \mid M \textbf{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}}$

*Theorem:* $L_{\mathrm{D}} \notin$ **RE**.

*Proof:* By contradiction; assume that $L_{\mathrm{D}} \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_{\mathrm{D}}$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_{\mathrm{D}} \text{ iff } \langle M \rangle \in \mathscr{L}(R) \tag{1}$$

From the definition of $L_{\mathrm{D}}$, we see that $\langle M \rangle \in L_{\mathrm{D}}$ iff $\langle M \rangle \notin \mathscr{L}(M)$.

# $L_{\text{D}}$ = { $\langle M \rangle$ | $M$ is a TM and $\langle M \rangle \notin \mathscr{L}(M)$ }

*Theorem:* $L_{\text{D}} \notin$ **RE**.

*Proof:* By contradiction; assume that $L_{\text{D}} \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_{\text{D}}$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_{\text{D}} \ \text{ iff } \ \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_{\text{D}}$, we see that $\langle M \rangle \in L_{\text{D}}$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \ \text{ iff } \ \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

## $L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_D \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_D$, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

We've replaced the left-hand side of this biconditional with an equivalent statement.

# $\boldsymbol{L_{\mathrm{D}} = \{ \langle M \rangle \mid M}$ is a TM and $\boldsymbol{\langle M \rangle \notin \mathscr{L}(M) \}}$

*Theorem:* $L_{\mathrm{D}} \notin$ **RE**.

*Proof:* By contradiction; assume that $L_{\mathrm{D}} \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_{\mathrm{D}}$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_{\mathrm{D}} \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \tag{1}$$

From the definition of $L_{\mathrm{D}}$, we see that $\langle M \rangle \in L_{\mathrm{D}}$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \tag{2}$$

Statement (2) holds for any TM $M$, so in particular it should hold when $M = R$.

## $L_D$ = { ⟨*M*⟩ | *M* is a TM and ⟨*M*⟩ ∉ $\mathscr{L}(M)$ }

*Theorem:* $L_D$ ∉ **RE**.

*Proof:* By contradiction; assume that $L_D$ ∈ **RE**. Then there must be some TM *R* such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if *M* is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_D$, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM *M*, so in particular it should hold when *M* = *R*.

A nice consequence of a universally-quantified statement is that it should work in all cases.

# $L_D = \{ \langle M \rangle \mid M$ is a TM and $\langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_D$, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM $M$, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

# $L_{\mathrm{D}} = \{ \langle \boldsymbol{M} \rangle \mid \boldsymbol{M}$ is a TM and $\langle \boldsymbol{M} \rangle \notin \mathscr{L}(\boldsymbol{M}) \}$

*Theorem:* $L_{\mathrm{D}} \notin$ **RE**.

*Proof:* By contradiction; assume that $L_{\mathrm{D}} \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_{\mathrm{D}}$.

Since $\mathscr{L}(R) = L_{D}$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_{\mathrm{D}} \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_{\mathrm{D}}$, we see that $\langle M \rangle \in L_{\mathrm{D}}$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM $M$, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \quad \text{iff} \quad \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible.

# $L_{\mathrm{D}} = \{ \langle M \rangle \mid M$ is a TM and $\langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_{\mathrm{D}} \notin \mathbf{RE}$.

*Proof:* By contradiction; assume that $L_{\mathrm{D}} \in \mathbf{RE}$. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_{\mathrm{D}}$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_{\mathrm{D}} \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_{\mathrm{D}}$, we see that $\langle M \rangle \in L_{\mathrm{D}}$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM $M$, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \quad \text{iff} \quad \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible. We have reached a contradiction, so our assumption must have been wrong.

# $L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of $L_D$, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM $M$, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible. We have reached a contradiction, so our assumption must have been wrong. Thus $L_D \notin$ **RE**.

## $L_D = \{ \langle M \rangle \mid M$ is a TM and $\langle M \rangle \notin \mathscr{L}(M) \}$

*Theorem:* $L_D \notin$ **RE**.

*Proof:* By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM $R$ such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if $M$ is any TM, then

$$\langle M \rangle \in L_D \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

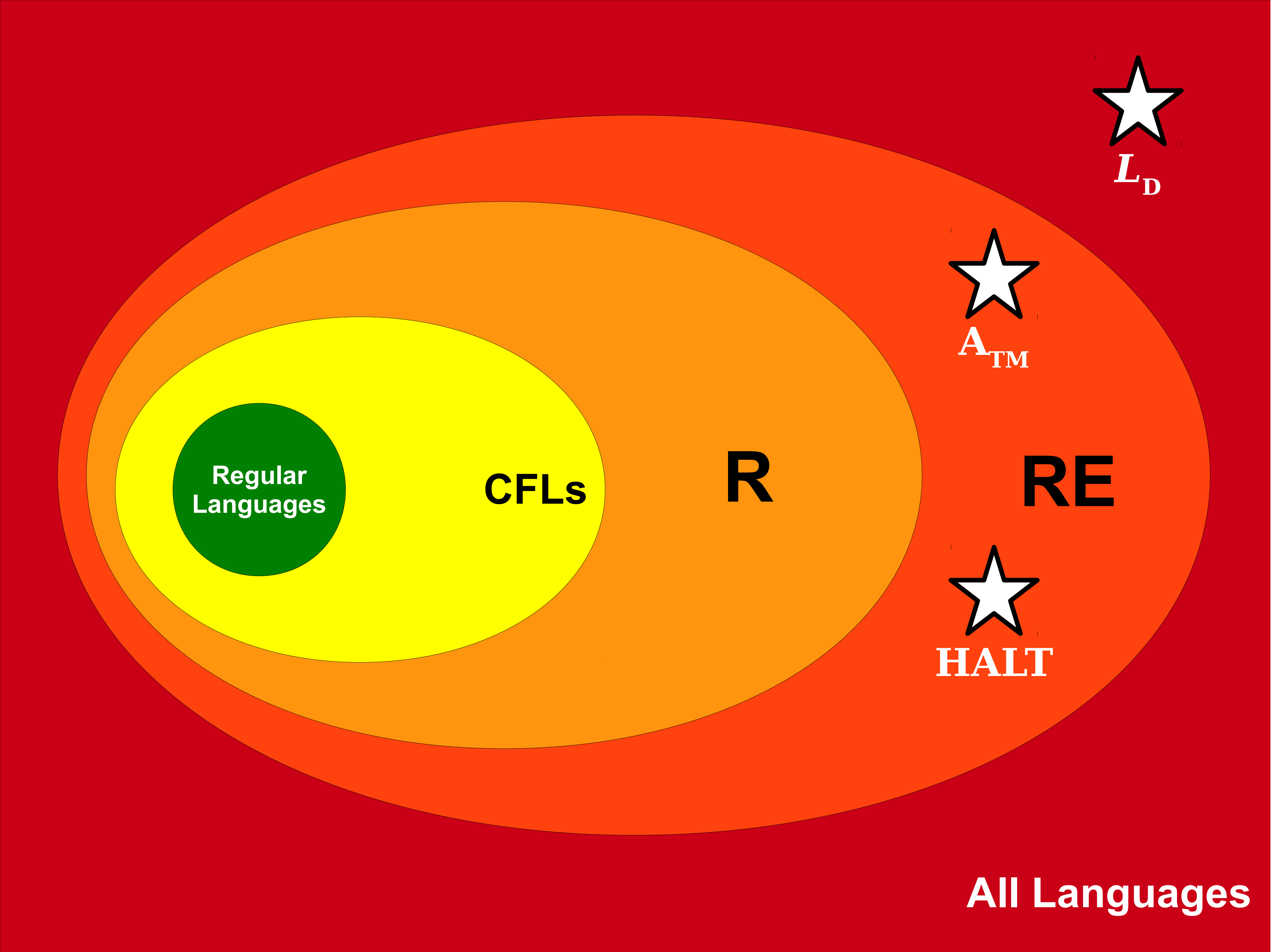
From the definition of $L_D$, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \quad \text{iff} \quad \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM $M$, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \quad \text{iff} \quad \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible. We have reached a contradiction, so our assumption must have been wrong. Thus $L_D \notin$ **RE**. ■


$L_D$
$A_{TM}$
Regular Languages
CFLs
R
RE
HALT
All Languages

# What This Means

- On a deeper philosophical level, the fact that non-**RE** languages exist supports the following claim:

  ***There are statements that are true but not provable.***

- Intuitively, given any non-**RE** language, there will be some string in the language that *cannot* be proven to be in the language.
- This result can be formalized as a result called ***Gödel's incompleteness theorem***, one of the most important mathematical results of all time.
- Want to learn more? Take Phil 152 or CS154!

# What This Means

- On a more philosophical note, you could interpret the previous result in the following way:

  ***There are inherent limits about what mathematics can teach us.***

- There's no automatic way to do math. There are true statements that we can't prove.
- That doesn't mean that mathematics is worthless. It just means that we need to temper our expectations about it.

# Time-Out for Announcements!

# Midterms

- You're done with midterms for this class! Woohoo!
- We're going to be grading the exam over the next couple of days. We're not sure if they'll be ready by Friday's lecture, but if they are, we'll be sure to release them then.
- Have questions in the meantime? Come talk to us!

# ESW Applications

- Stanford's chapter of Engineers for a Sustainable World are taking applications for their Global Engineering Opportunities program.
- Excellent program that has a real impact. Highly recommended!
- There's an accompanying Winter-Spring course series that goes along with this (CEE177X/CEE177S), and I've heard great things about it.
- Project descriptions are available here. Applications are available online here.
- Applications are due on November 27.

# Your Questions

“Last lecture, we learned there is a difference between what is true and what we can discover to be true. How has that implication changed your perspective on your own life, outside of a math or computer science context?”

These results we have are just pure math. We can interpret them however we'd like. One interpretation is that math and CS are provably incapable of always giving us the truth. Another is that some statements might be neither true nor false.

I think this helps keep a healthy perspective on how we define and arrive at truth. Math can't tell us everything, and I personally think that no one system is always going to have all the answers. That's a good thing to keep in mind in a world where everyone will claim they have all the answers.

“Bump from last time: best piece of advice you ever received? ☺”

I'm still thinking about it! Nothing is coming to mind right now…

“What 3 important things would you list on the back of Midterm 2?”

“In the spirit of Thanksgiving, what are you most thankful for?”

1. Family / Friends
2. Freedom, safety, and stability
3. Curiosity

Back to CS103!

# Finding Non-**RE** Languages

# Self-Reference and **RE**

- Self-reference is the main technique we'll use to find non-**RE** languages.
- However, things are a bit more complicated when finding non-**RE** languages than finding non-**R** languages.
- Why?
  - When talking about deciders, we assume we have a magic subroutine that always produces the answer we want.
  - **RE** is the class of problems with recognizers and verifiers. These are much harder to reason about.

# The Unhalting Problem

- Consider this problem:

**Given a TM *M* and a string *w*, does *M* loop on *w*?**

- As a formal language:

$$\boldsymbol{LOOP = \{ \langle M, w \rangle \mid M \text{ is a TM that loops on } w \}}$$

- How hard of a problem is this to solve?

# The Unhalting Problem

- Intuitively, would we expect this language to belong to class **R**?
  - ***No:*** The only general way to figure out what a TM will do is to run it.
  - Can formalize this by building a program that asks if it will loop and does the opposite.
- Intuitively, would we expect this language to belong to class **RE**?
  - ***Remember:*** The only general way to find out what a TM is going to do is to run it.
  - If you ran a TM and you didn't see it accept or reject, that doesn't mean that it's looping! You may just need more time.
  - How would you convince someone that it was looping?

# Self-Reference and Verifiers

- Assume for the sake of contradiction that *LOOP* ∈ **RE**.
- This means that there's a verifier for *LOOP*.
- In software, that would be a function like this one:

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate)
```

- This function is our verifier:
  - If `program` loops on `input`, then there is some choice of `certificate` that causes this function to return true.
  - If `program` halts on `input`, this function always returns false, regardless of what `certificate` is.
  - The function always returns a value.

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
    /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
    /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

... this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
    /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

... this program halts on its input?

*There is no certificate that proves it loops.*

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

... this program halts on its input?

*There is no certificate that proves it loops.*

So the program loops on its input!

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

*There is a certificate that proves it loops.*

So the program halts on its input!

… this program halts on its input?

*There is no certificate that proves it loops.*

So the program loops on its input!

***Theorem:*** $LOOP \notin \mathbf{RE}$.

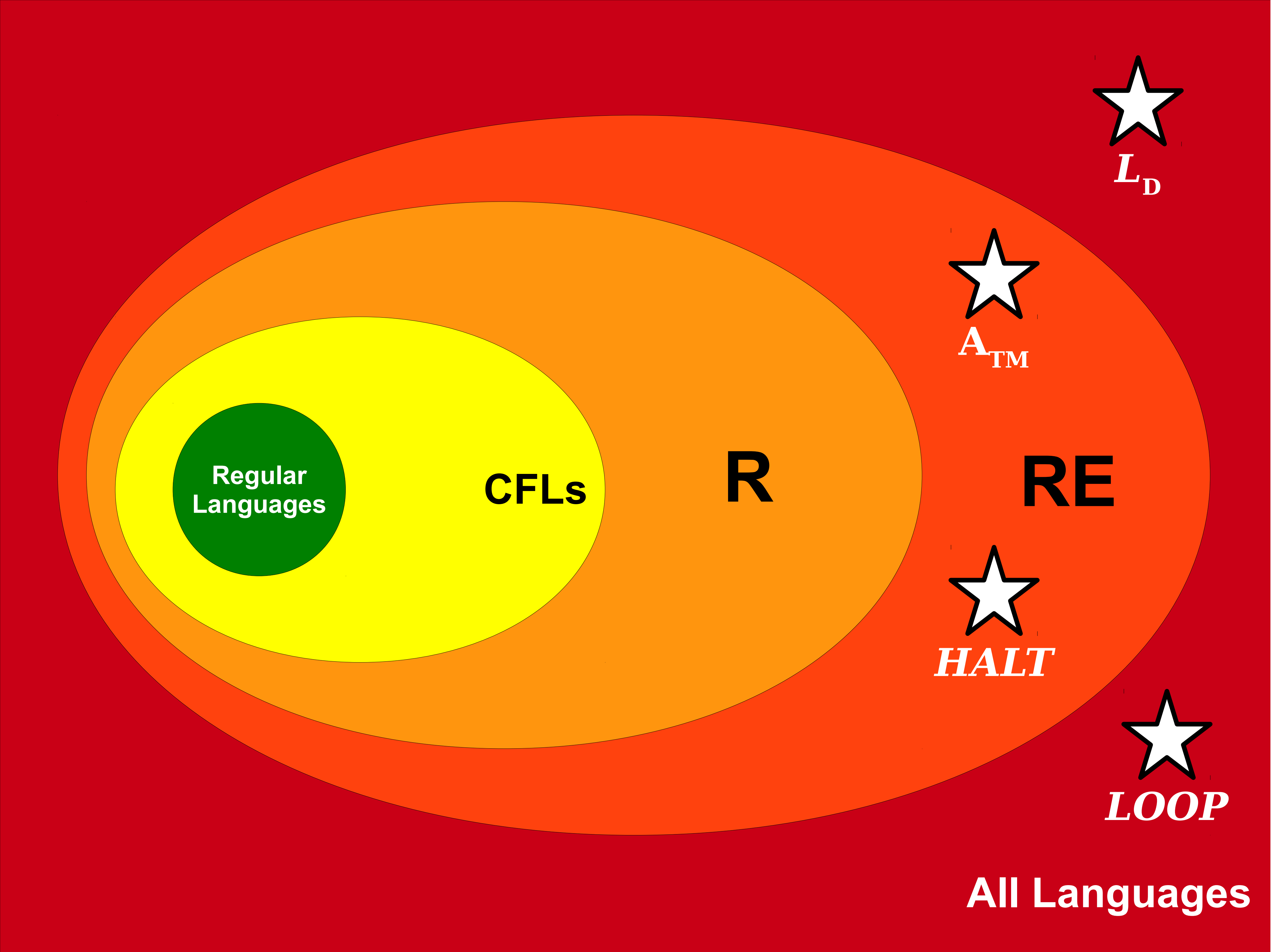
***Proof:*** By contradiction; assume that $LOOP \in \mathbf{RE}$. Then there is some verifier $V$ for $LOOP$. This verifier has the property that if $M$ is a TM that loops on some string $w$, there is a certificate $c$ such that $V$ accepts $\langle M, w, c\rangle$, and if $M$ halts on $w$, $V$ will never accept $\langle M, w, c\rangle$ for any certificate $c$.

Given this, we could then construct the following TM:

```
$M$ = "On input $w$:
          Have $M$ obtain its own description, $\langle M\rangle$.
          For all strings $c$:
              If $V$ accepts $\langle M, w, c\rangle$, accept.
```

Choose any string $w$ and trace through the execution of the machine. If $V$ ever accepts $\langle M, w, c\rangle$, we are guaranteed that $M$ loops on $w$, but in this case we find that $M$ accepts $w$, a contradiction. If $V$ never accepts $\langle M, w, c\rangle$, then we are guaranteed that $M$ halts on $w$, but in this case we find that $M$ loops infinitely on $w$, a contradiction.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $LOOP \notin \mathbf{RE}$. ■


Regular Languages
CFLs
R
RE
$L_D$
$A_{TM}$
*HALT*
*LOOP*
All Languages

# *LOOP* ∉ **RE**

- The fact that *LOOP* ∉ **RE** gives us a powerful intuition:

***There is no general way to prove that a TM will loop on a particular input.***

- This is a really useful intuition going forward – it will help you get a better sense for when a problem is likely to be unrecognizable.

# Self-Reference and **RE**

- The proof template for showing undecidability via self-reference is the following:
  - Build a machine that asks what it is about to do.
  - Based on the answer, have the machine do the exact opposite.
- The proof template for showing *unrecognizability* via self-reference is the following:
  - Build a machine that tries to *prove* it will do something.
  - If it *proves* that it will, have it do the opposite.
  - If it can't *prove* that it will, it will loop infinitely. Design the machine so that looping infinitely causes it to behave incorrectly.

# Another Non-**RE** Language

- Let's revisit our secure voting problem:

**Given a TM *M*, is the language of *M***
**{ *w* ∈ {r, d}* | *w* has more r's than d's }?**

- We know that this language isn't decidable (we proved that a while back).
- Is it recognizable?
- ***Good intuition:*** If you *knew* that a TM was a secure voting machine, what could you do to prove it to me?

# Secure Voting

- Suppose that the secure voting problem is recognizable. Then we could write a function

  ```
  bool imConvincedIsSecure(string program,
                           string certificate)
  ```

  that would accept as input a program and a certificate, then would check whether that certificate “proves” that the program is a secure voting machine.
- We're going to use this to build a self-referential program that causes problems:
  - If there is a certificate that makes it a secure voting machine, we're going to make it behave in a way that makes it *not* a secure voting machine.
  - If there are no certificates that make it behave like a secure voting machine, we're going to make it behave in a way that makes it a secure voting machine.

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if…

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

## What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }
   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();
   if (countRs(input) > countDs(input)) {
      accept();
   }
   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

## What will we accept?

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more r's than d's.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }
   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

## What will we accept?

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

## What will we accept?

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more
more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

## What will we accept?

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

Everything else.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more r's than d's.

Everything else.

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

... this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

... this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

... this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

... this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();
   if (countRs(input) > countDs(input)) {
      accept();
   }
   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }
    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if...

... this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

... this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }
   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (countRs(input) > countDs(input)) {
        accept();
    }

    for (i = 0 to infinity) {
        for (each string c of length i) {
            if (imConvincedIsSecure(me, c)) {
                accept();
            }
        }
    }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

## What will we accept?

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

### What will we accept?

All strings that have more more r's than d's.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

   for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

Nothing else.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (countRs(input) > countDs(input)) {
        accept();
    }

    for (i = 0 to infinity) {
        for (each string c of length i) {
            if (imConvincedIsSecure(me, c)) {
                accept();
            }
        }
    }
}
```

**What will we accept?**

All strings that have more more `r`'s than `d`'s.

Nothing else.

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

So the program is a secure voting machine!

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if…

… this program is a secure voting machine?

*There is a certificate that proves it's secure.*

So the program isn't a secure voting machine!

… this program is not a secure voting machine?

*There is no certificate that proves it loops.*

So the program is a secure voting machine!

# Secure Voting

- The argument we've just made shows that, in general, there's no way to prove that a voting system is secure!
- What options do we have?
  - Restrict the sorts of programs we can write so that they're not as powerful as Turing machines.
  - Have a trusted person write the voting software.
  - Don't allow the program to be so complicated that it's hard to reason about.
  - Rely on the opinions of experts to validate the software.
  - Not trust a computer with elections.
- ***Just because this problem is impossible doesn't mean that we shouldn't try to solve it.*** We just need to recalibrate our expectations.

# Where We Stand

- We've just done a crazy, whirlwind tour of computability theory:
  - ***The Church-Turing thesis*** tells us that TMs give us a mechanism for studying computation in the abstract.
  - ***Universal computers*** – computers as we know them – are not just a stroke of luck. The existence of the universal TM ensures that such computers must exist.
  - ***Self-reference*** is an inherent consequence of computational power.
  - ***Undecidable problems*** exist partially as a consequence of the above and indicate that there are statements whose truth can't be determined by computational processes.
  - ***Unrecognizable problems*** also exist partially through self-reference and indicate that there are limits to mathematical proof.

# Where We've Been

- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.

# Where We're Going

- The class **P** represents problems that can be solved *efficiently* by a computer.
- The class **NP** represents problems where “yes” answers can be verified *efficiently* by a computer.