

Unsolvable Problems

Part Two

Outline for Today

- ***Recap from Last Time***
 - Where are we, again?
- ***A Different Perspective on RE***
 - What exactly does “recognizability” mean?
- ***Verifiers***
 - A new approach to problem-solving.
- ***Beyond RE***
 - Monstrously hard problems!

Recap from Last Time

Self-Referential Programs

- ***Claim:*** Any program can be augmented to include a method called `mySource()` that returns a string representation of its source code.
- ***Theorem:*** It is possible to build Turing machines that get their own encodings and perform arbitrary computations on them.

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!

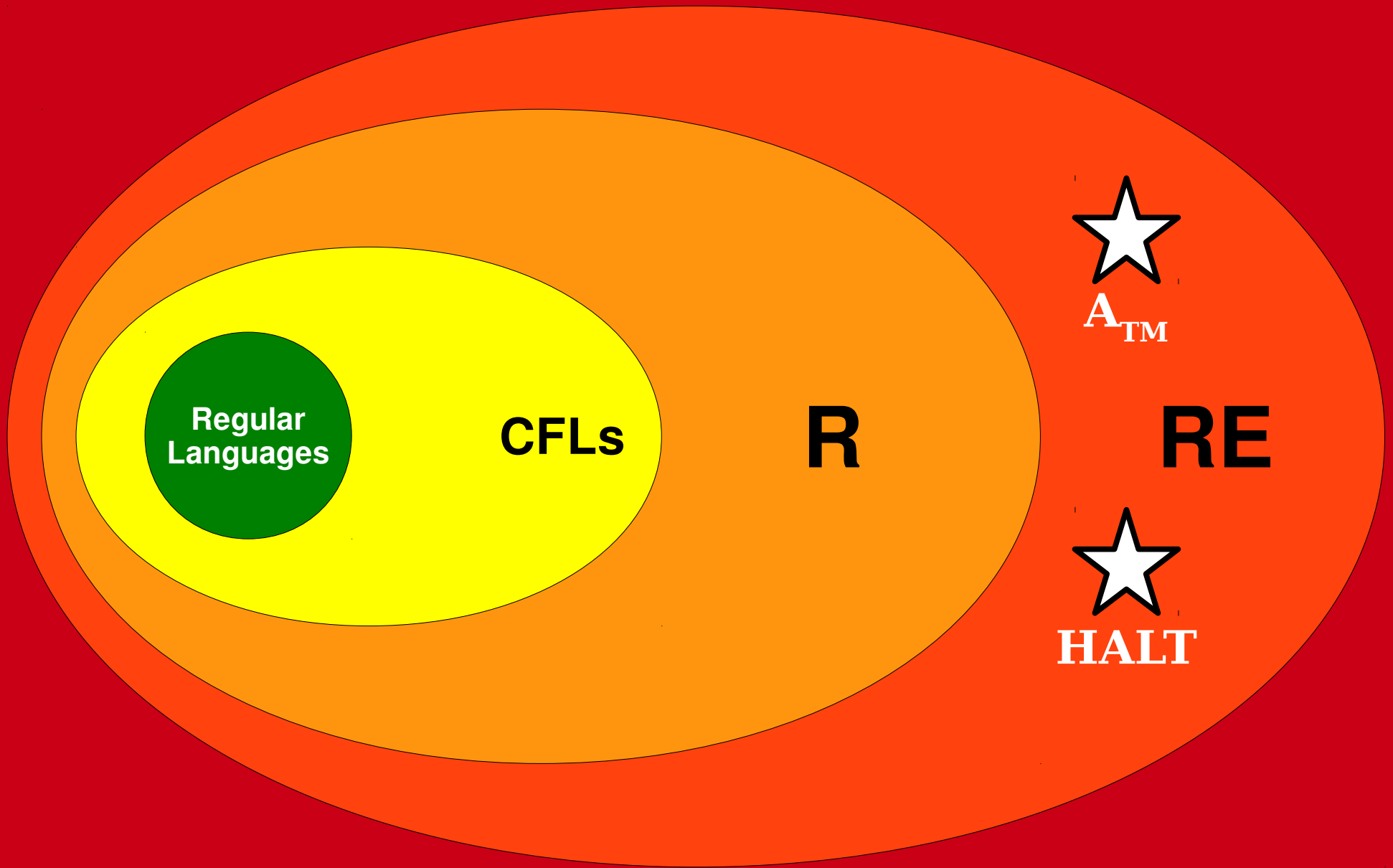
... this program doesn't accept its input?
It accepts the input!

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!



All Languages

New Stuff!

Beyond **R** and **RE**

Beyond **R** and **RE**

- We've now seen how to use self-reference as a tool for showing undecidability (finding languages not in **R**).
- We still have not broken out of **RE** yet, though.
- To do so, we will need to build up a better intuition for the class **RE**.

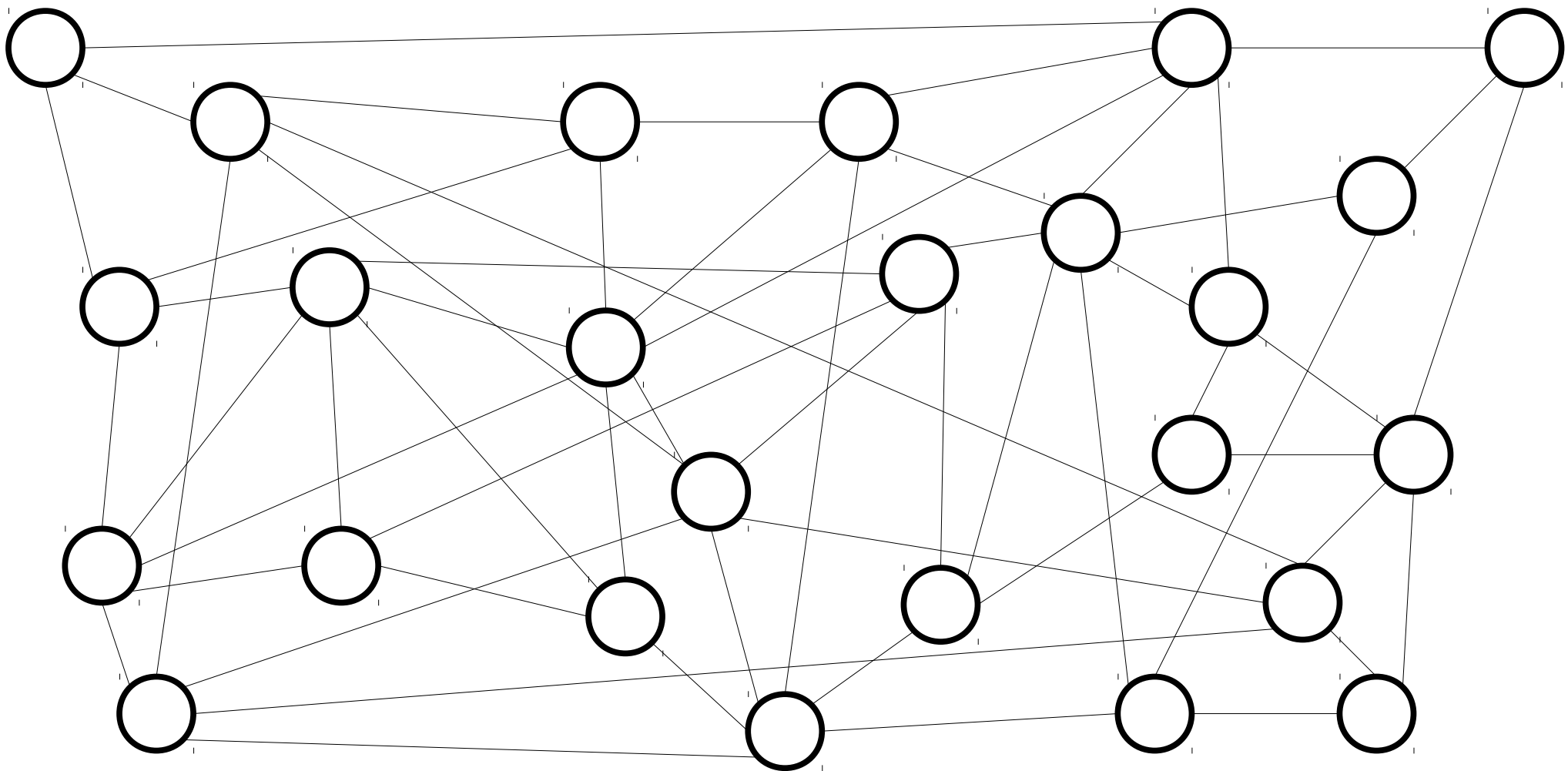
What exactly is the class **RE**?

RE, Formally

- Recall that the class **RE** is the class of all recognizable languages:

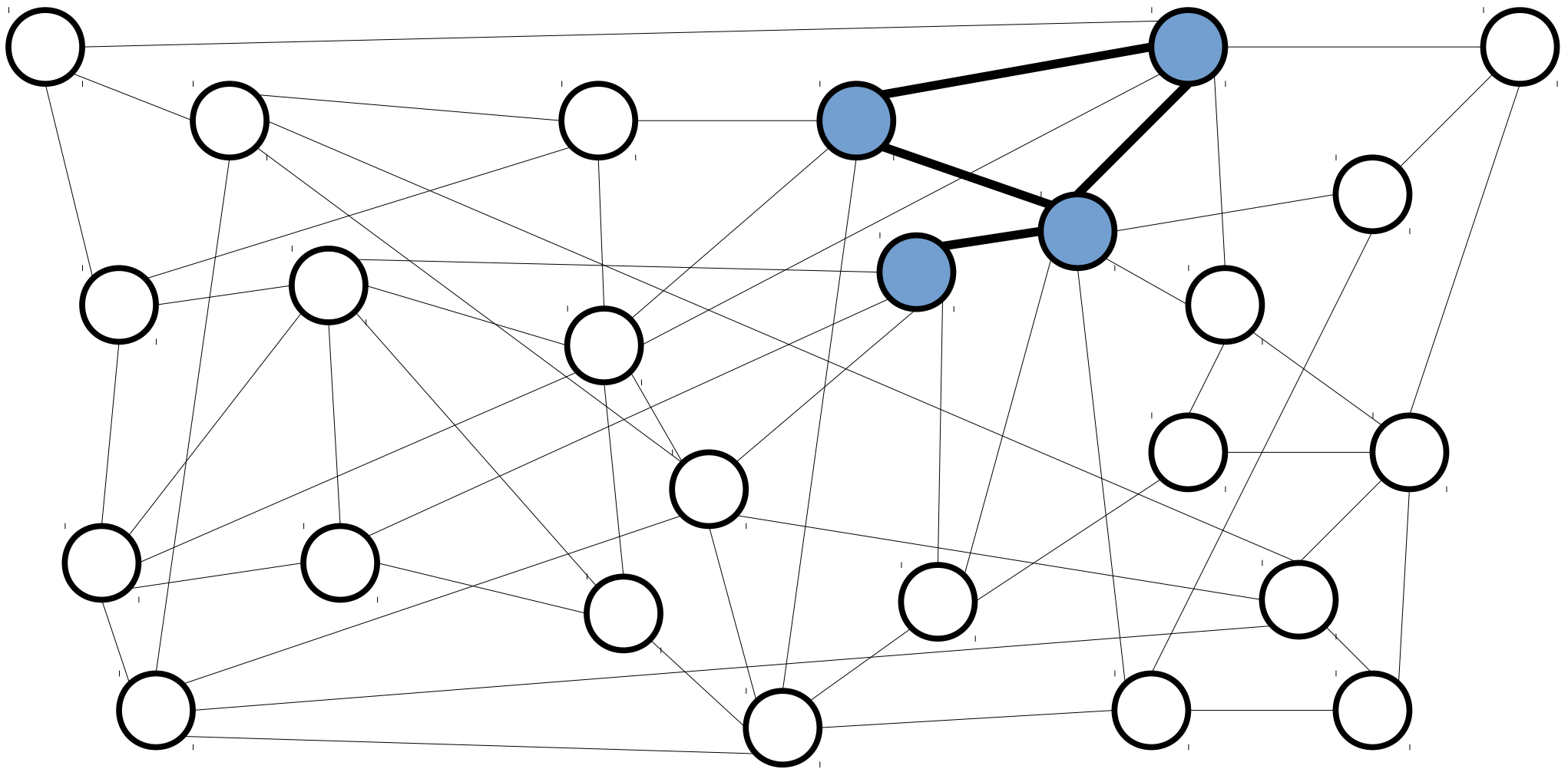
$$\mathbf{RE} = \{ L \mid \text{there is a TM } M \text{ where } \mathcal{L}(M) = L \}$$

- Since $\mathbf{R} \neq \mathbf{RE}$, there is no general way to “solve” problems in the class **RE**, if by “solve” you mean “make a computer program that can always tell you the correct answer.”
- So what exactly *are* the sorts of languages in **RE**?



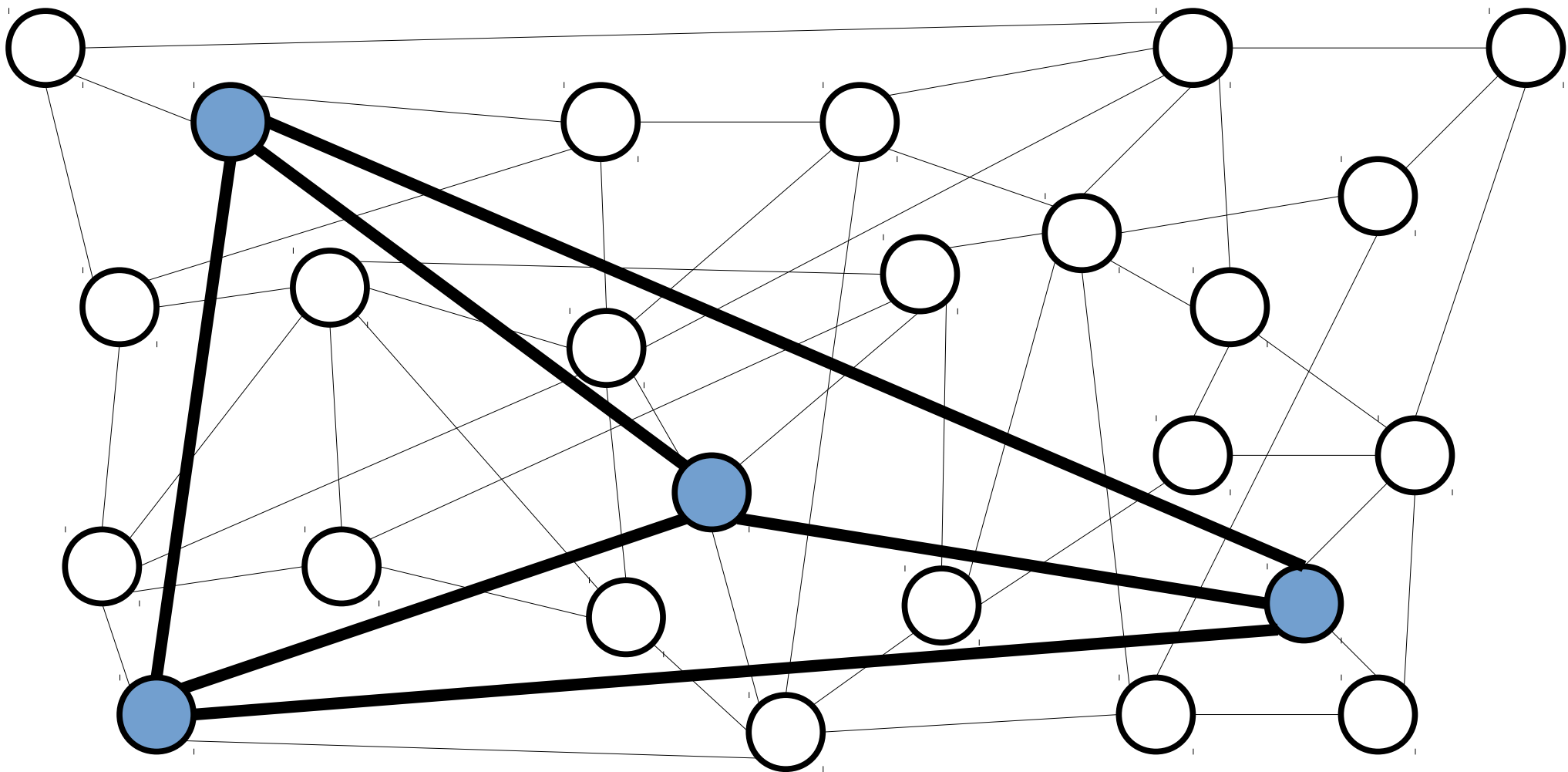
Does this graph contain a 4-clique?

Answer at [Pollevo.com/cs103](https://www.pollevo.com/cs103) or
text **CS103** to **22333** once to join, then **Y** or **N**.



Does this graph contain a 4-clique?

Answer at [Pollev.com/cs103](https://pollev.com/cs103) or
text **CS103** to **22333** once to join, then **Y** or **N**.



Does this graph contain a 4-clique?

Answer at [Pollevo.com/cs103](https://www.pollevo.com/cs103) or
text **CS103** to **22333** once to join, then **Y** or **N**.

Key Intuition:

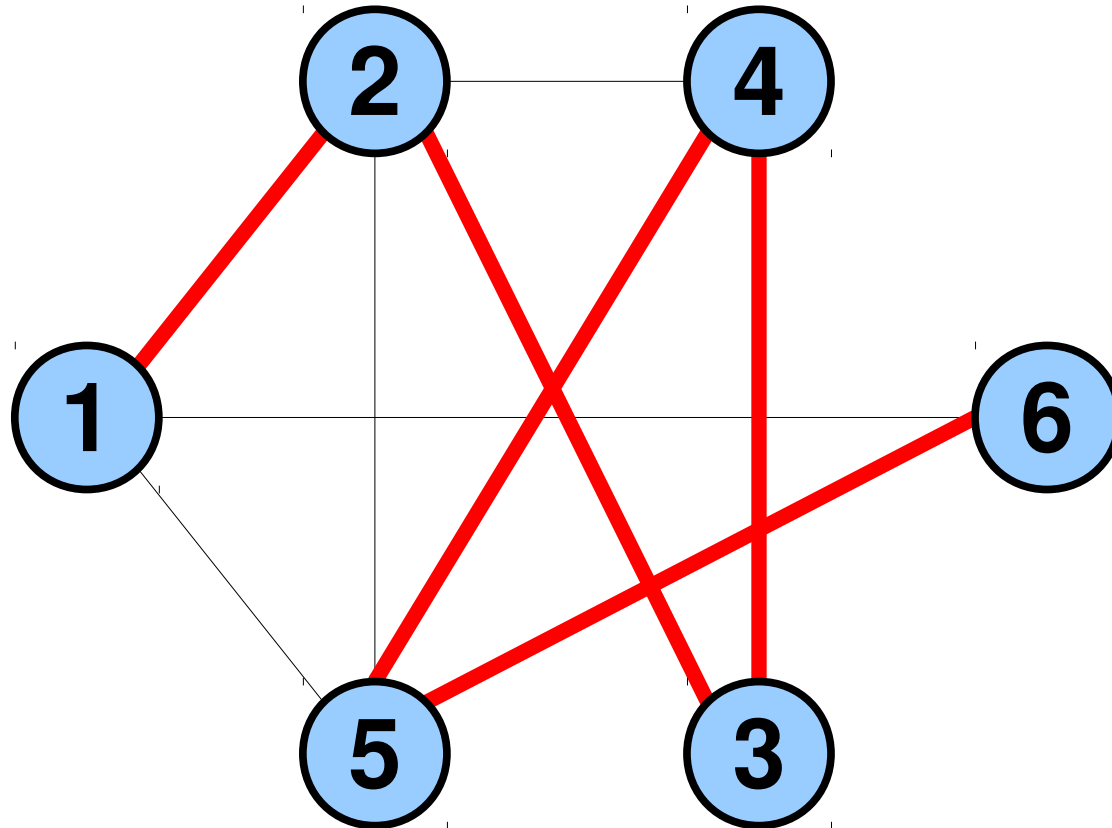
A language L is in **RE** if, for any string w , if you are *convinced* that $w \in L$, there is some way you could prove that to someone else.

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



Does this graph have a ***Hamiltonian path*** (a simple path that passes through every node exactly once?)

Verification

11

Try running fourteen steps of the Hailstone sequence.

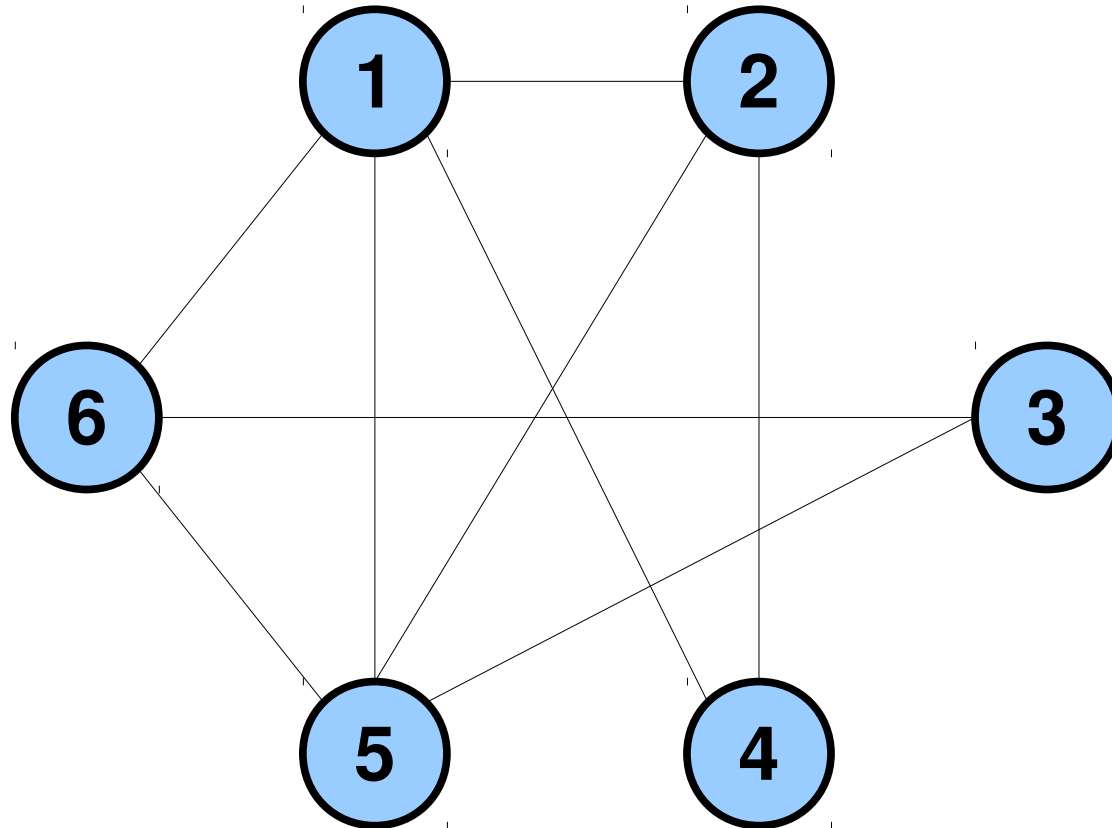
Does the hailstone sequence
terminate for this number?

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



Does this graph have a ***Hamiltonian path*** (a simple path that passes through every node exactly once?)

Verification

11

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

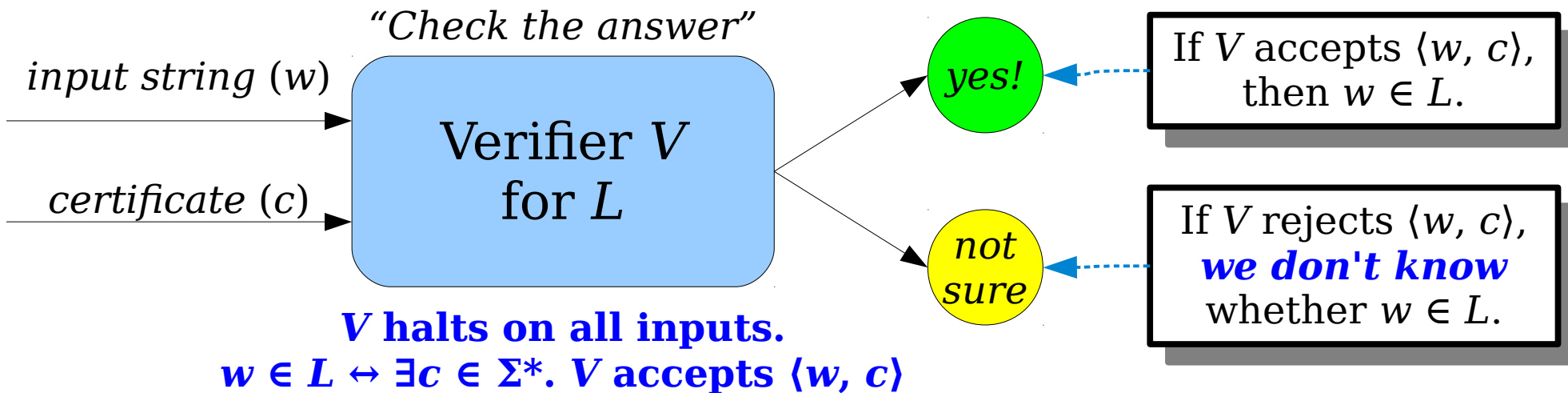
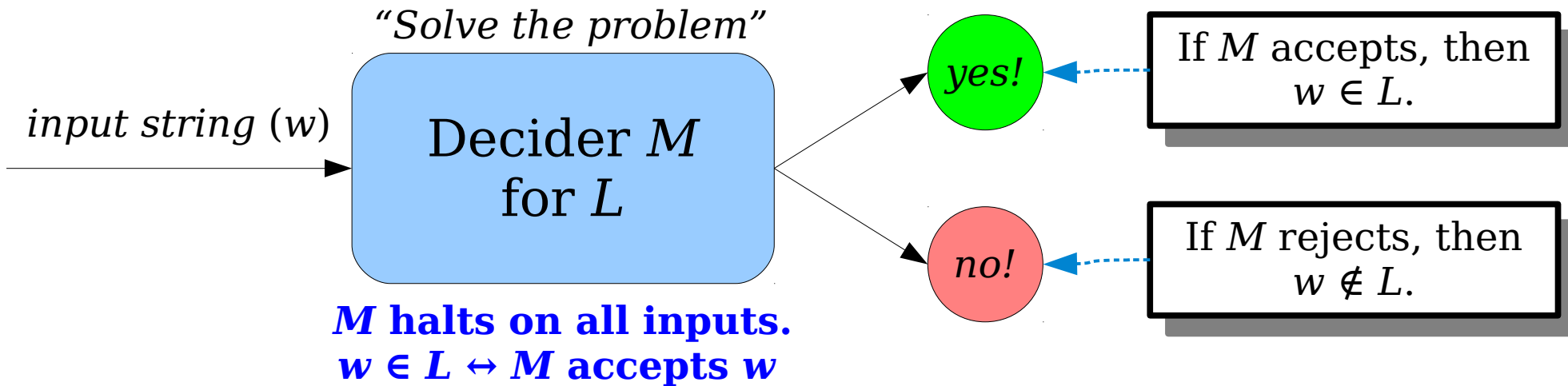
Verification

- In each of the preceding cases, we were given some problem and some evidence supporting the claim that the answer is “yes.”
- Given the correct evidence, we can be certain that the answer is indeed “yes.”
- Given incorrect evidence, we aren't sure whether the answer is “yes.”
 - Maybe there's *no* evidence saying that the answer is “yes,” or maybe there is some evidence, but just not the evidence we were given.
- Let's formalize this idea.

Verifiers

- A **verifier** for a language L is a TM V with the following properties:
 - V halts on all inputs.
 - For any string $w \in \Sigma^*$, the following is true:
$$w \in L \leftrightarrow \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle$$
- A string c where V accepts $\langle w, c \rangle$ is called a **certificate** for w .
- Intuitively, what does this mean?

Deciders and Verifiers



Verifiers

- A **verifier** for a language L is a TM V with the following properties:
 - V halts on all inputs.
 - For any string $w \in \Sigma^*$, the following is true:
$$w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle$$
- 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.

Verifiers

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

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

- Some notes about V :
 - Notice that c is existentially quantified. Any string $w \in L$ must have at least one c that causes V to accept, and possibly more.
 - V is required to halt, so given any potential certificate c for w , you can check whether the certificate is correct.

Verifiers

- A **verifier** for a language L is a TM V with the following properties:
 - V halts on all inputs.
 - For any string $w \in \Sigma^*$, the following is true:
$$w \in L \leftrightarrow \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle$$
- Some notes about V :
 - Notice that $\mathcal{L}(V) \neq L$. (*Good question: what is $\mathcal{L}(V)$?*)
 - The job of V is just to check certificates, not to decide membership in L .

Verifiers

- A **verifier** for a language L is a TM V with the following properties:
 - V halts on all inputs.
 - For any string $w \in \Sigma^*$, the following is true:
$$w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle$$
- Some notes about V :
 - Although this formal definition works with a string c , remember that c can be an encoding of some other object.
 - In practice, c will likely just be “some other auxiliary data that helps you out.”

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

- Let's see how to build a verifier for L .

Verification

11

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Some Verifiers

- Let L be the following language:

$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$

```
bool checkHailstone(int n, int c) {  
    for (int i = 0; i < c; i++) {  
        if (n % 2 == 0) n /= 2;  
        else n = 3*n + 1;  
    }  
    return n == 1;  
}
```

- Do you see why $\langle n \rangle \in L$ iff there is some c such that `checkHailstone(n, c)` returns true?
- Do you see why `checkHailstone` always halts?

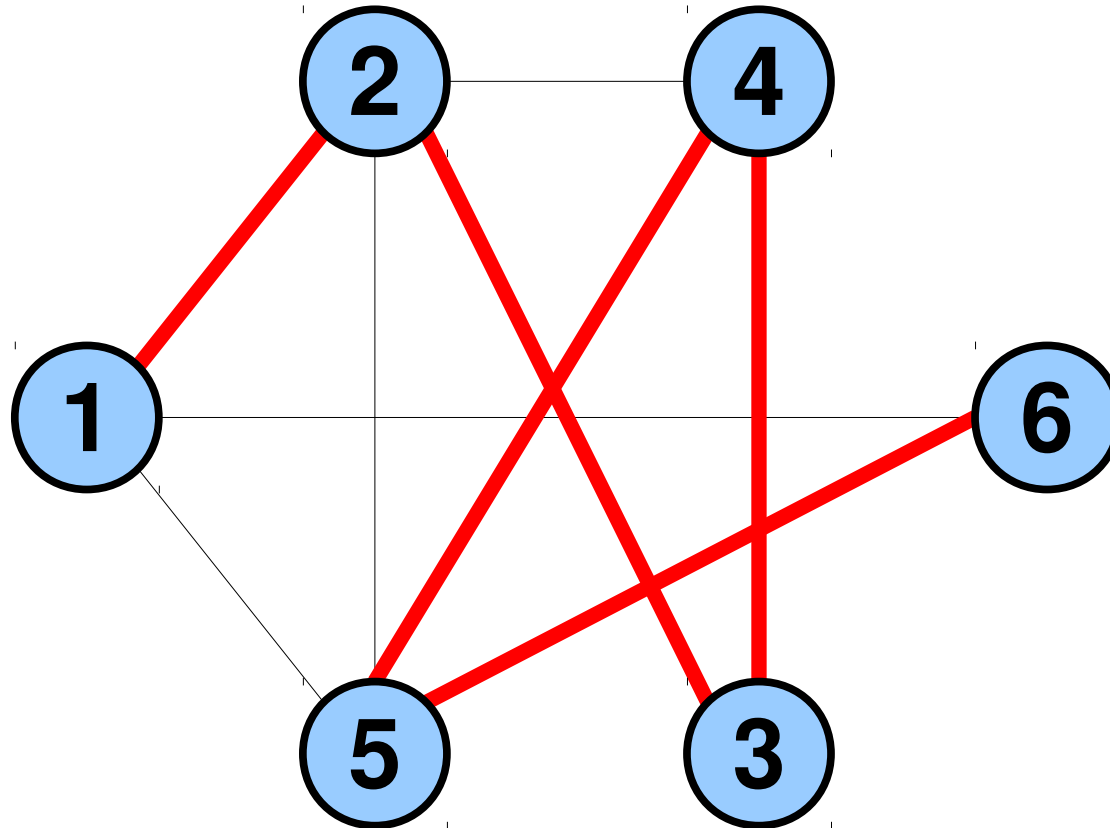
Some Verifiers

- Let L be the following language:

$$L = \{ \langle G \rangle \mid G \text{ is a graph and } G \text{ has a Hamiltonian path } \}$$

- (A Hamiltonian path is a simple path that visits every node in the graph.)
- Let's see how to build a verifier for L .

Verification



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

Some Verifiers

- Let L be the following language:

$L = \{ \langle G \rangle \mid G \text{ is a graph with a Hamiltonian path} \}$

```
bool checkHamiltonian(Graph G, vector<Node> c) {  
    if (c.size() != G.numNodes()) return false;  
    if (containsDuplicate(c)) return false;  
  
    for (size_t i = 0; i < c.size() - 1; i++) {  
        if (!G.hasEdge(c[i], c[i+1])) return false;  
    }  
    return true;  
}
```

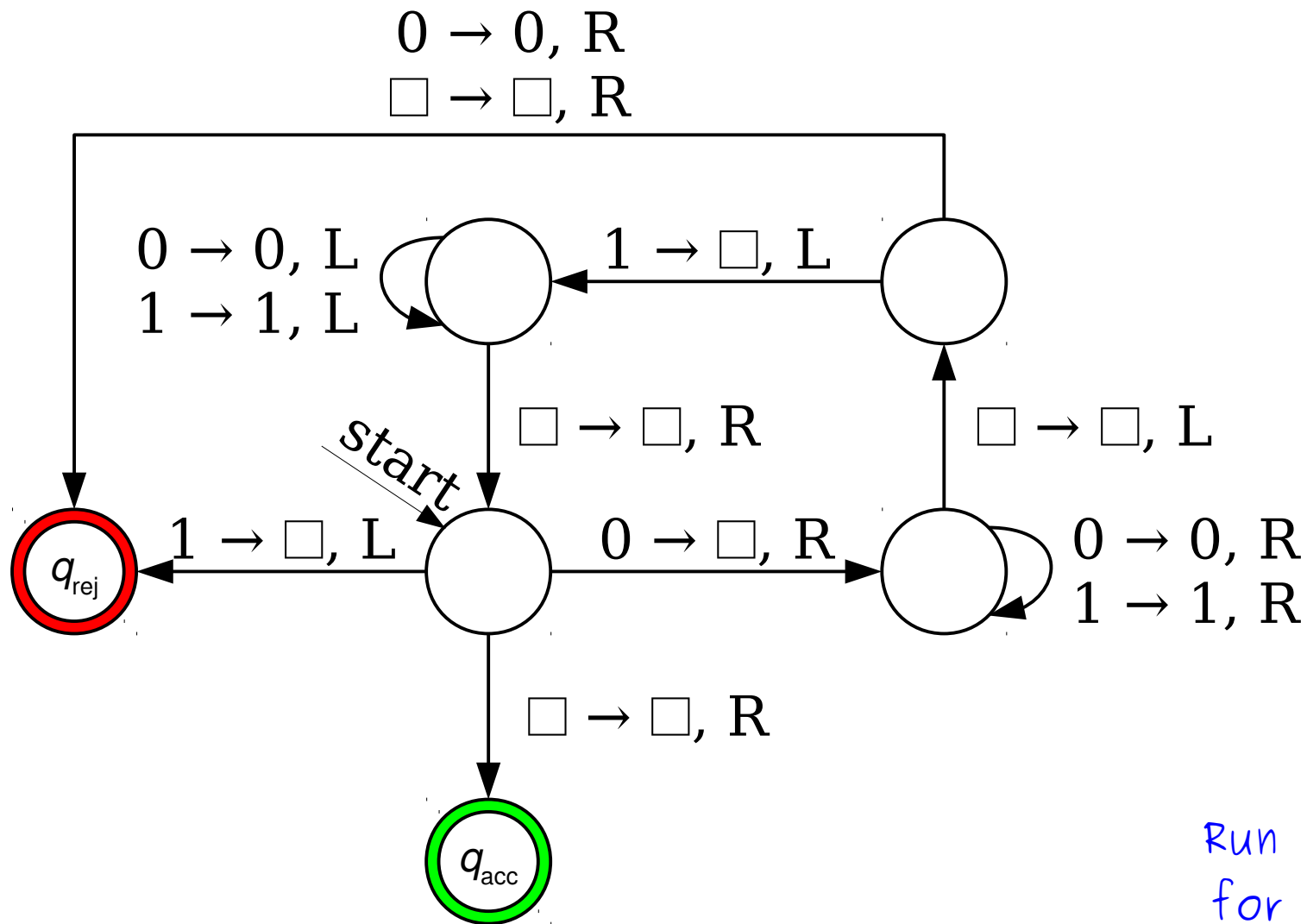
- Do you see why $\langle G \rangle \in L$ iff there is a c where `checkHamiltonian(G, c)` returns true?
- Do you see why `checkHamiltonian` always halts?

Some Verifiers

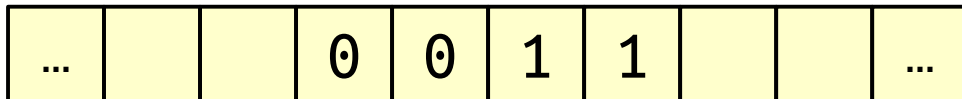
- Consider A_{TM} :

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

- This is a *canonical* example of an undecidable language. There's no way, in general, to tell whether a TM M will accept a string w .
- Although this language is undecidable, it's an **RE** language, and it's possible to build a verifier for it!



Run this TM
for fifteen
steps.



Some Verifiers

- Consider A_{TM} :

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

```
bool checkWillAccept(TM M, string w, int c) {  
    set up a simulation of M running on w;  
    for (int i = 0; i < c; i++) {  
        simulate the next step of M running on w;  
    }  
    return whether M is in an accepting state;  
}
```

- Do you see why M accepts w iff there is some c such that `checkWillAccept(M, w, c)` returns true?
- Do you see why `checkWillAccept` always halts?

What languages are verifiable?

Let V be a verifier for a language L . Consider the following function given in pseudocode:

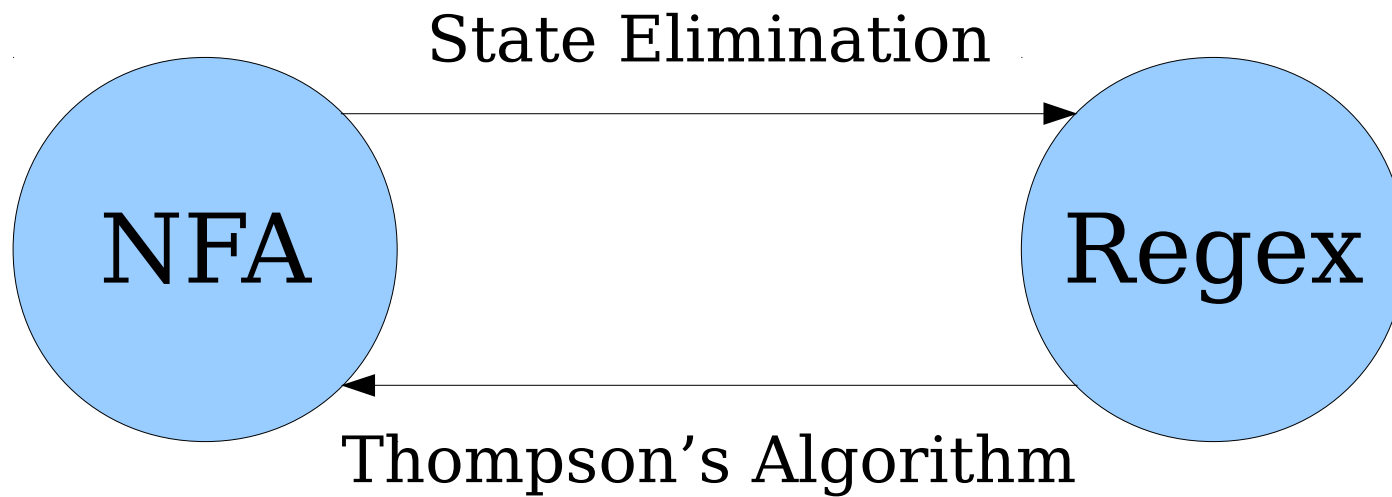
```
bool mysteryFunction(string w) {  
    int i = 0;  
    while (true) {  
        for (each string c of length i) {  
            if (V accepts  $\langle w, c \rangle$ ) return true;  
        }  
        i++;  
    }  
}
```

What set of strings does `mysteryFunction` return **true** on?

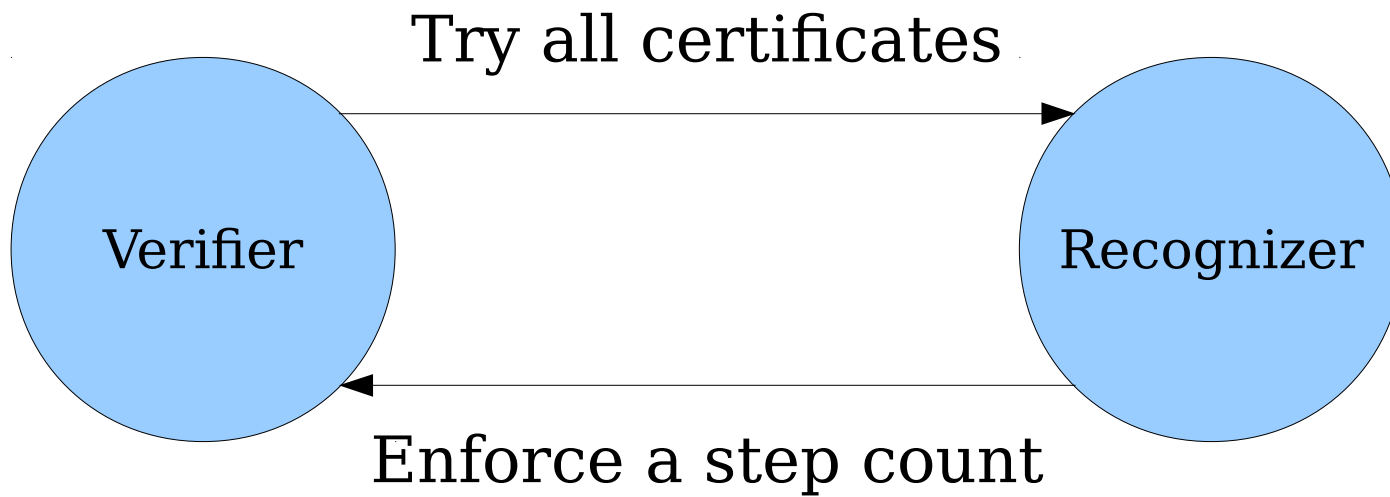
Answer at [PollEv.com/cs103](https://www.pollEv.com/cs103) or
text **CS103** to **22333** once to join, then **your answer**.

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

Where We've Been



Where We're Going

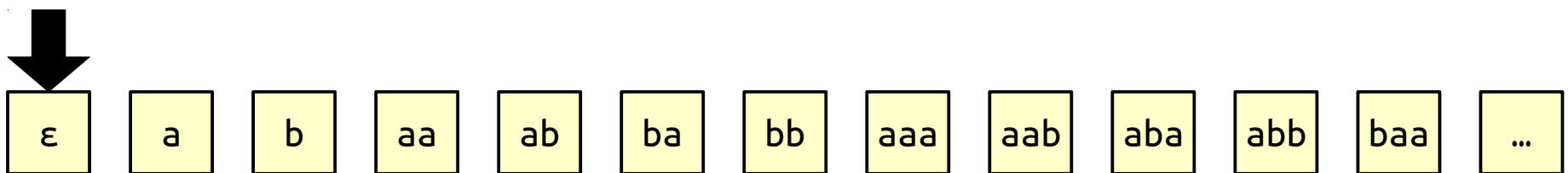
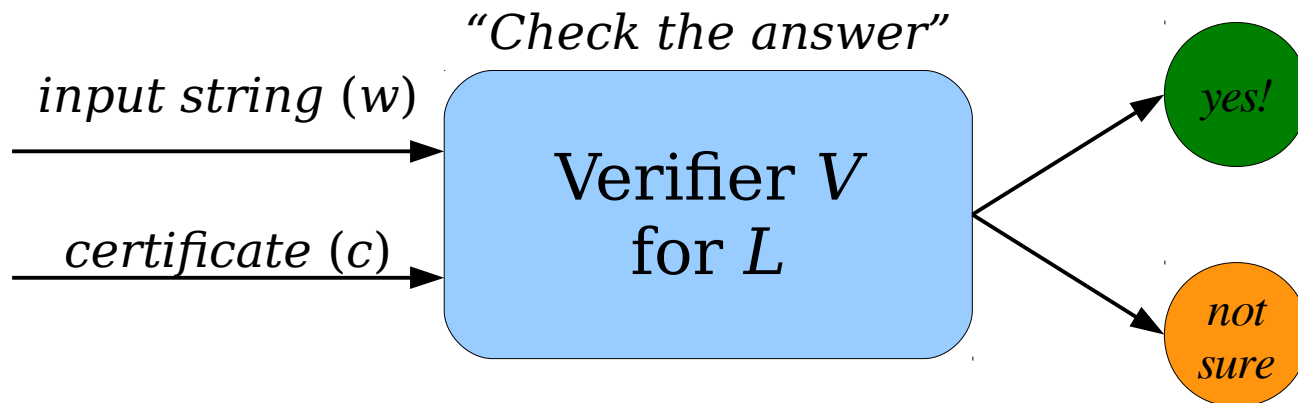


Verifiers and **RE**

- **Theorem:** If there is a verifier V for a language L , then $L \in \mathbf{RE}$.
- **Proof goal:** Given a verifier V for a language L , find a way to construct a recognizer M for L .

Verifiers and **RE**

- **Theorem:** If there is a verifier V for a language L , then $L \in \mathbf{RE}$.
- **Proof goal:** Given a verifier V for a language L , find a way to construct a recognizer M for L .



Verifiers and **RE**

- **Theorem:** If V is a verifier for L , then $L \in \mathbf{RE}$.
- **Proof sketch:** Consider the following program:

```
bool isInL(string w) {  
    int i = 0;  
    while (true) {  
        for (each string c of length i) {  
            if (V accepts ⟨w, c⟩) return true;  
        }  
        i++;  
    }  
}
```

If $w \in L$, there is some $c \in \Sigma^*$ where V accepts $\langle w, c \rangle$. The function `isInL` tries all possible strings as certificate, so it will eventually find c (or some other certificate), see V accept $\langle w, c \rangle$, then return true. Conversely, if `isInL(w)` returns true, then there was some string c such that V accepted $\langle w, c \rangle$, so $w \in L$. ■

Verifiers and **RE**

- **Theorem:** If $L \in \mathbf{RE}$, 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 challenges:
 - A recognizer M is not required to halt on all inputs. A verifier V must always halt.
 - A recognizer M takes in one single input. A verifier V takes in two inputs.
- We'll need to find a way of reconciling these requirements.

Recall: If M is a recognizer for a language L , then M accepts w iff $w \in L$.

Key insight: If M accepts a string w , it always does so in a finite number of steps.

Idea: Adapt the verifier for A_{TM} into a more general construction that turns any recognizer into a verifier by running it for a fixed number of steps.

Verifiers and RE

- **Theorem:** If $L \in \mathbf{RE}$, then there is a verifier for L .
- **Proof sketch:** Consider the following program:

```
bool checkIsInL(string w, int c) {  
    set up a simulation of M running on w;  
    for (int i = 0; i < c; i++) {  
        simulate the next step of M running on w;  
    }  
    return whether M is in an accepting state;  
}
```

Notice that `checkIsInL` always halts, since each step takes only finite time to complete. Next, notice that if there is a c where `checkIsInL(w, c)` returns true, then M accepted w after running for c steps, so $w \in L$. Conversely, if $w \in L$, then M accepts w after some number of steps (call that number c). Then `checkIsInL(w, c)` will run M on w for c steps, watch M accept w , then return true. ■

RE and Proofs

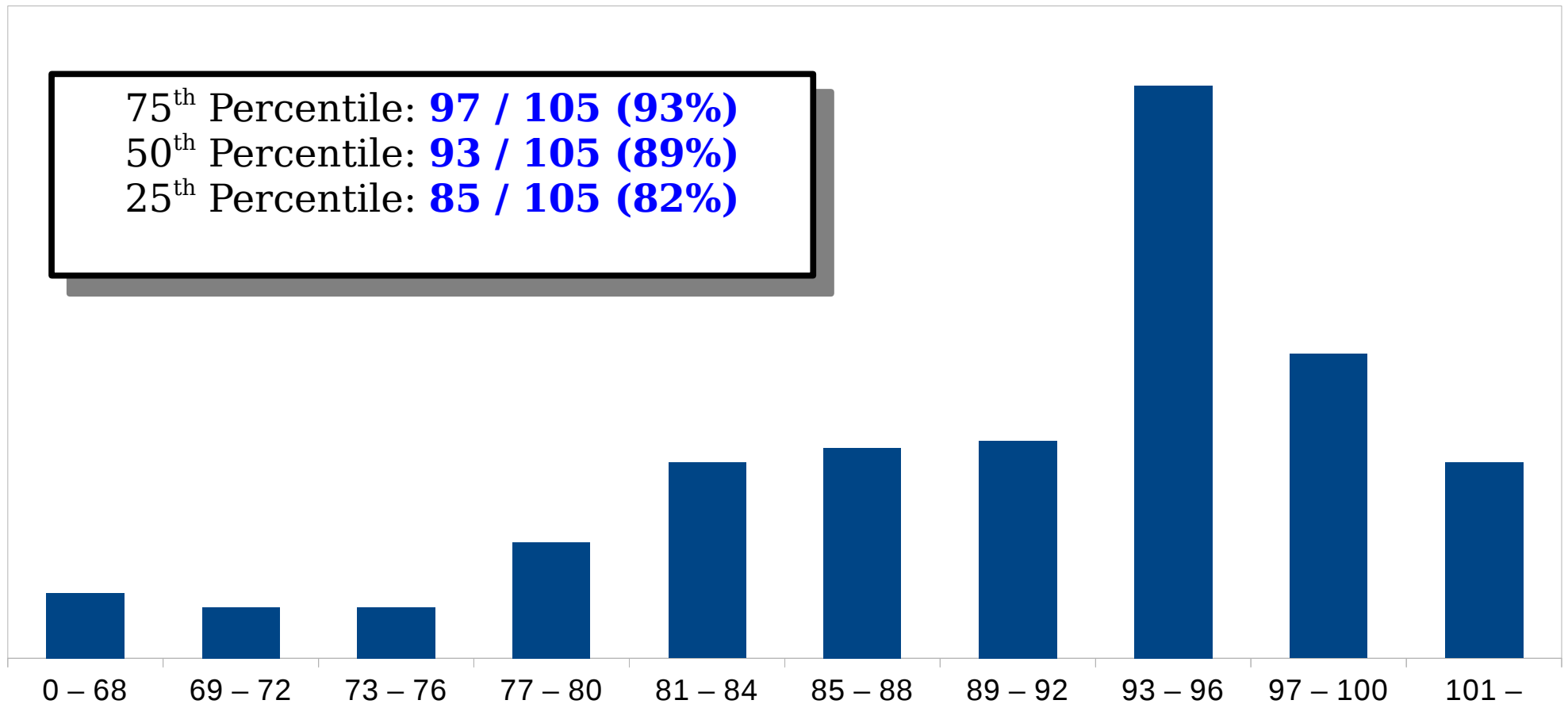
- Verifiers and recognizers give two different perspectives on the “proof” intuition for **RE**.
- Verifiers are explicitly built to check proofs that strings are in the language.
 - If you know that some string w belongs to the language and you have the proof of it, you can convince someone else that $w \in L$.
- 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!

Time-Out for Announcements!

Problem Set Seven Graded



Problem Set Nine

- Problem Set Eight was due today at 2:30PM.
 - You *can* use late days here to extend the deadline as far as Sunday at 2:30PM, but we don't recommend this.
- Problem Set Nine goes out today. It's due next Friday at 2:30PM.
 - Play around with the limits of **R** and **RE** languages – the upper extent of computation!
 - See how everything fits together!
- Due to university policies, ***no late submissions will be accepted for PS9***. Please budget at least two hours before the deadline to submit the assignment.

The Last Two Guides

- We've posted two final guides to the course website:
 - The ***Guide to Self-Reference***, which talks about proofs of undecidability via self-reference.
 - The ***Guide to the Lava Diagram***, which provides an intuition for how different classes of languages relate to one another.
- Give these a read – there's a ton of useful information in there!

Final Exam Logistics

- Our final exam is Monday, March 19th from 3:30PM – 6:30PM, location Hewlett 200 & 201 (no special last name assignments).
 - Sorry about how soon that is – the registrar picked this time, not us. If we had a choice, it would be on the last day of finals week.
- The exam is cumulative. You're responsible for topics from PS1 – PS9 and all of the lectures.
- As with the midterms, the exam is closed-book, closed-computer, and limited-note. You can bring one double-sided sheet of 8.5" × 11" notes with you to the exam, decorated any way you'd like.
- Students with OAE accommodations: if we don't yet have your OAE letter, please send it to us ASAP.

Preparing for the Exam

- We've posted **six** practice final exams, with solutions, to the course website.
- These exams are essentially the final exams we've given out in the last six quarters, with a few tweaks and modifications.
- Practice Final 1 and Practice Final 6 are the two most recent exams and should give you the best indicator of the expected topic coverage.
- And don't forget that Extra Practice Problems 3 is available online. After today's lecture, you know enough to take on any of those questions, including the starred ones.

Back to CS103!

Finding Non-**RE** Languages

Finding Non-**RE** Languages

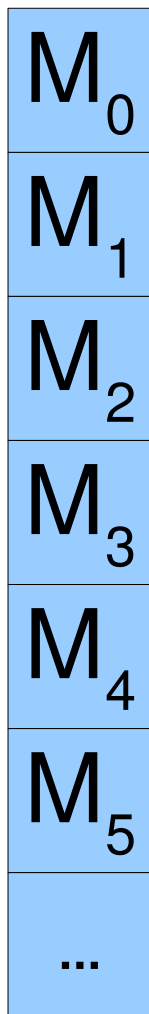
- Right now, we know that non-**RE** languages exist, but we have no idea what they look like.
- How might we find one?

Languages, TMs, and TM Encodings

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

$$\mathcal{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

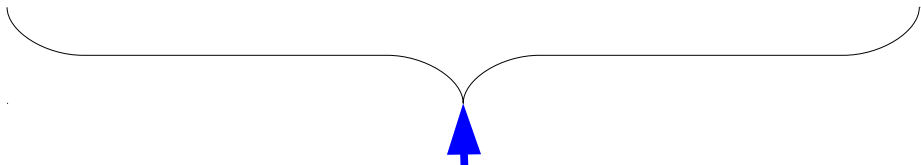
- Some of the strings in this set might be descriptions of TMs.
- What happens if we list off all Turing machines, looking at how those TMs behave given other TMs as input?



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	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	...
...

What are we going to do next?

Answer at [Pollev.com/cs103](https://pollev.com/cs103) or
text **CS103** to **22333** once to join, then **your answer**.

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$...
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	...
...

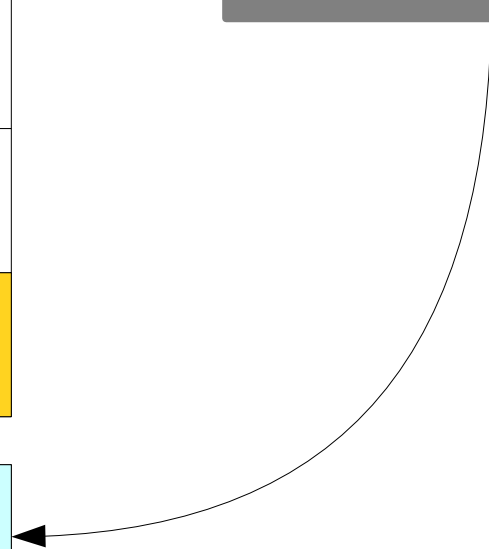
Flip all "accept" to "no" and vice-versa

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 ...

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	...
...

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

No	No	No	Acc	No	Acc	...
----	----	----	-----	----	-----	-----

Diagonalization Revisited

- The ***diagonalization language***, which we denote L_D , is defined as

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

- That is, L_D is the set of descriptions of Turing machines that do not accept themselves.

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

Theorem: $L_D \notin \mathbf{RE}$.

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

Let M be an arbitrary TM. Since $\mathcal{L}(R) = L_D$, we know that

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

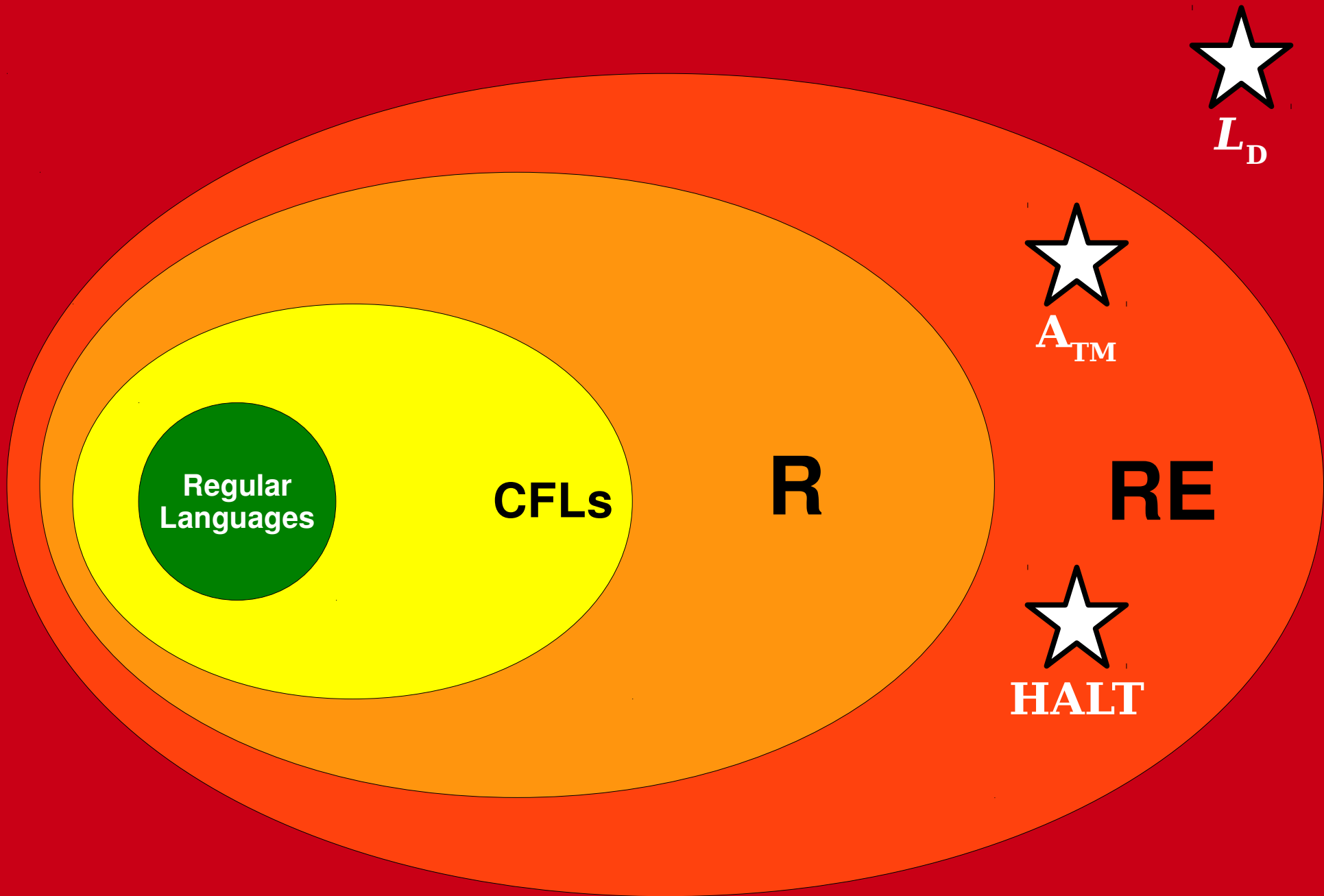
From the definition of L_D , we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathcal{L}(M)$. Combining this with statement (1) tells us that

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

Since our choice of M was arbitrary, we see that statement (2) holds for any TM M . In particular, this means that statement (2) holds for the TM R , which tells us that

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

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



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:

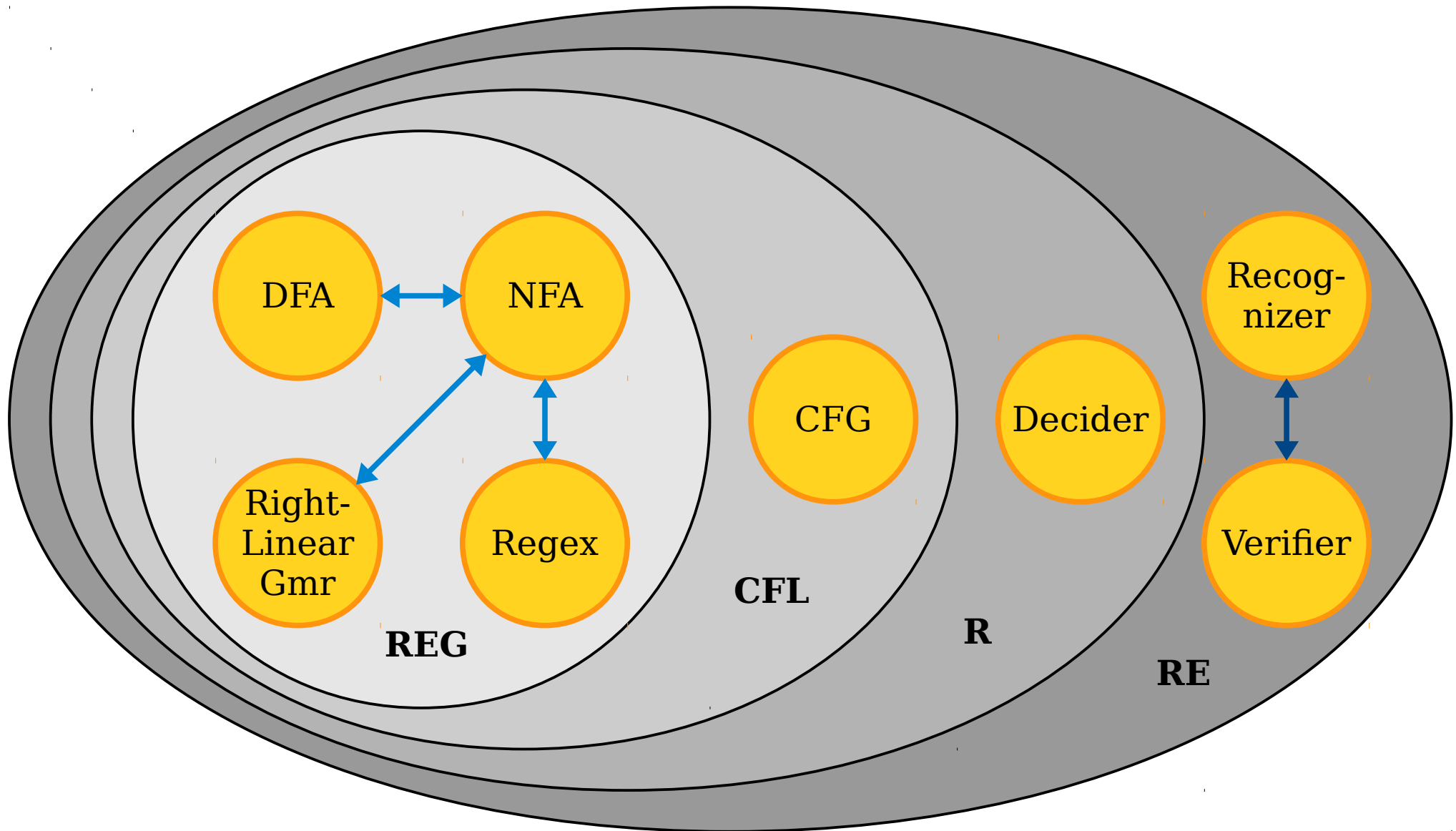
There are inherent limits about what mathematics can teach us.

- 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.

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*** are out there and can be discovered via diagonalization. They imply there are limits to mathematical proof.

The Big Picture



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.

Next Time

- ***Introduction to Complexity Theory***
 - Not all decidable problems are created equal!
- ***The Classes P and NP***
 - Two fundamental and important complexity classes.
- ***The $P \stackrel{?}{=} NP$ Question***
 - A literal million-dollar question!