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

A beautiful example of an impossible problem.

Recap from Last Time

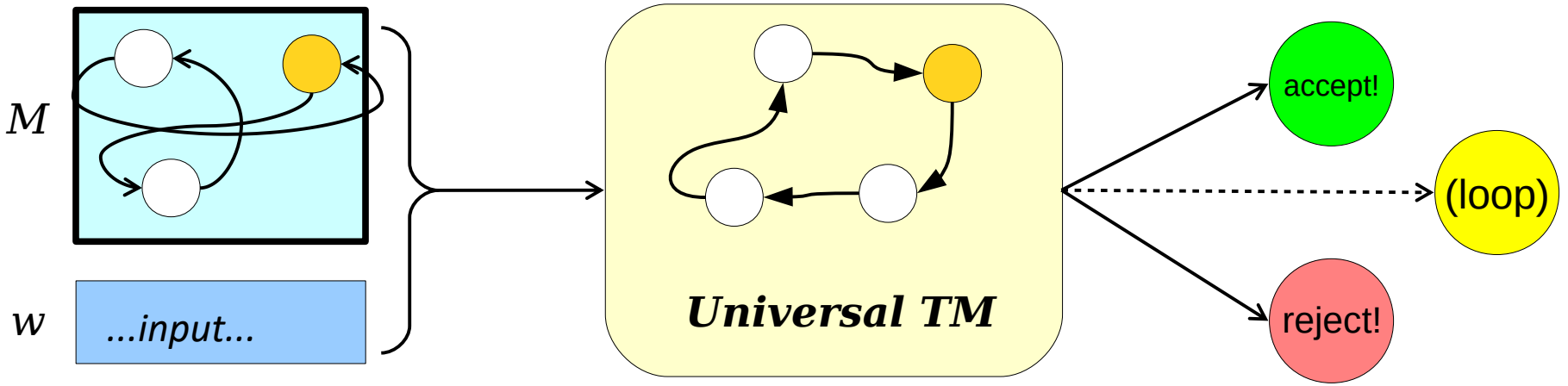
The Universal Turing Machine

- ***Theorem (Turing, 1936)***: There is a Turing machine U_{TM} called the ***universal Turing machine*** that, when run on an input of the form $\langle M, w \rangle$, where M is a Turing machine and w is a string, simulates M running on w and does whatever M does on w (accepts, rejects, or loops).
- The observable behavior of U_{TM} is the following:
  - If M accepts w, then U_{TM} accepts $\langle M, w \rangle$.
  - If M rejects w, then U_{TM} rejects $\langle M, w \rangle$.
  - If M loops on w, then U_{TM} loops on $\langle M, w \rangle$.

M does to w

what

U_{TM} does to $\langle M, w \rangle$.



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 it possible to build Turing machines that get their own encodings and perform arbitrary computations on them.

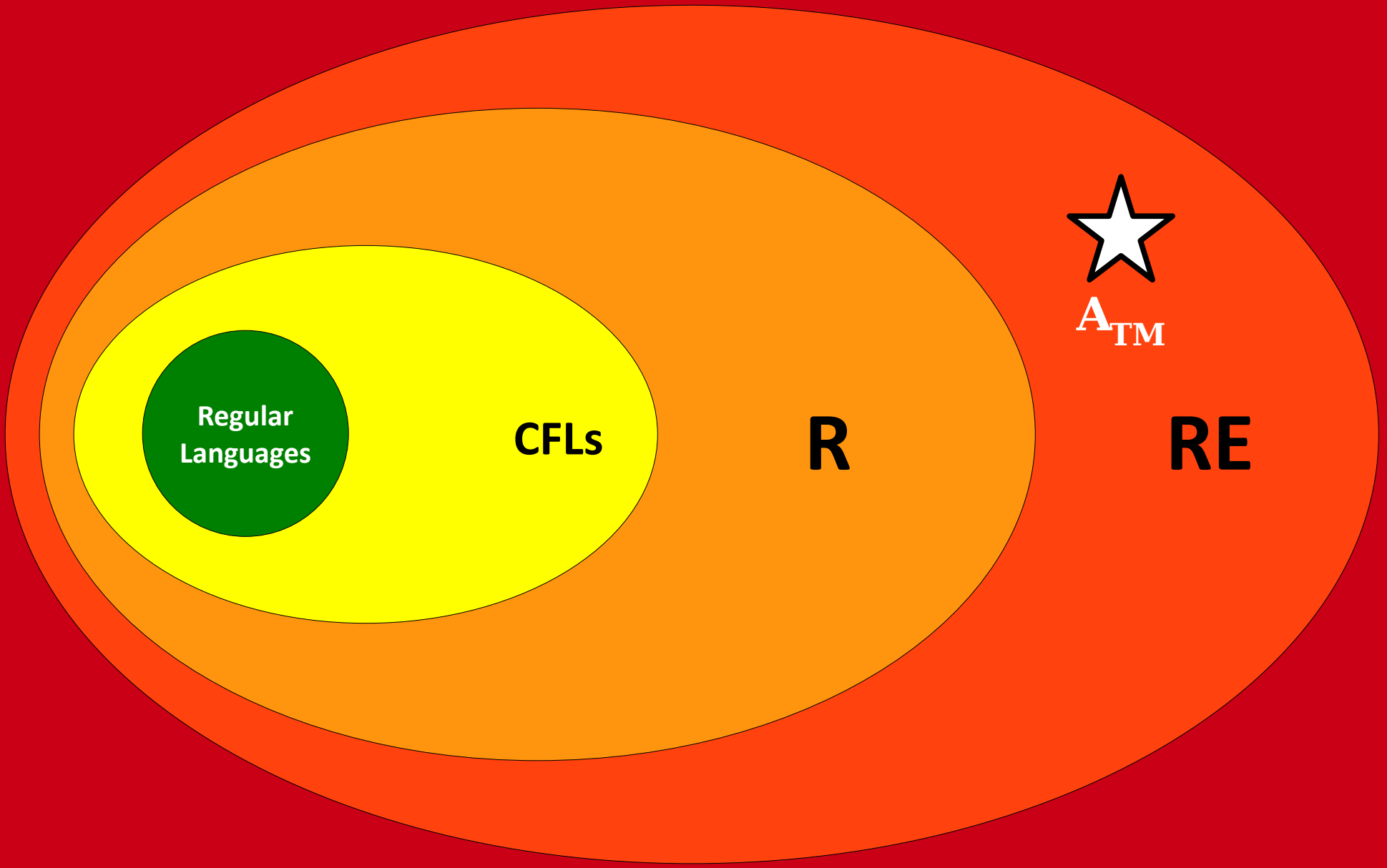
What does this program do?

```
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!


Regular Languages
CFLs
R
RE
A_{TM}
All Languages

New Stuff!

More Impossibility Results

The Halting Problem

The most famous undecidable problem is the ***halting problem***, which asks:

Given a TM *M* and a string *w*,
will *M* halt when run on *w*?

As a formal language, this problem would be expressed as

***HALT* = { ⟨*M*, *w*⟩ | *M* is a TM that halts on *w* }**

This is an **RE** language. *(We'll see why later.)*

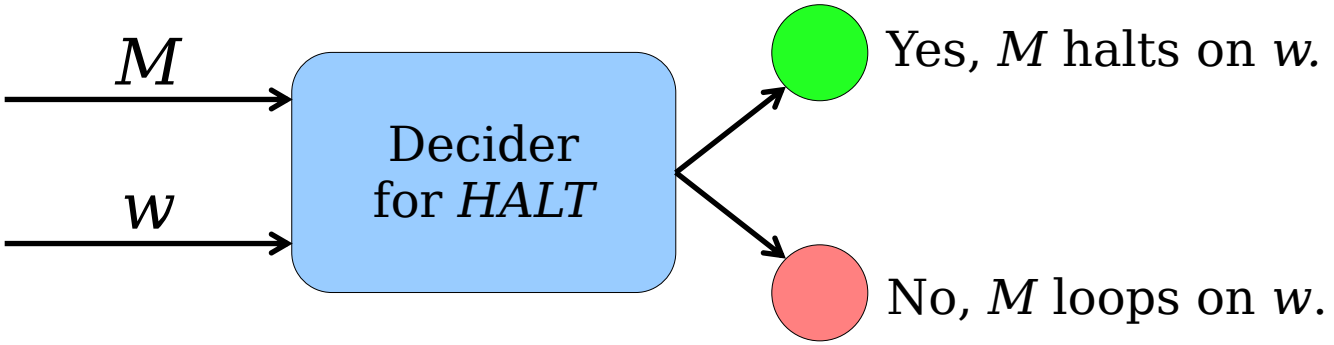
How do we know that it's undecidable?

Claim: A decider for *HALT* is a self-defeating object. It therefore doesn't exist.

A Decider for *HALT*

Let's suppose that, somehow, we managed to build a decider for $HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$.

Schematically, that decider would look like this:



We could represent this decider in software as a method

```
bool willHalt(string program, string input);
```

that takes as input a program and a string, then returns whether that program will halt on that string.

What does this program do?

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

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 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

… this program halts on that 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

… this program halts on that 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

… this program halts on that 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

… this program halts on that 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

… this program halts on that 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

It loops on 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if…

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

… this program loops on this 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if…

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

… this program loops on this 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if…

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

… this program loops on this 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

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

… this program loops on this 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 infinite
                }
        } else {
                accept();
        }
}
```

Imagine running this program on some input. What happens if...

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

… this program loops on this input?
It halts on 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 infinite
                }
        } else {
                accept();
        }
}
```

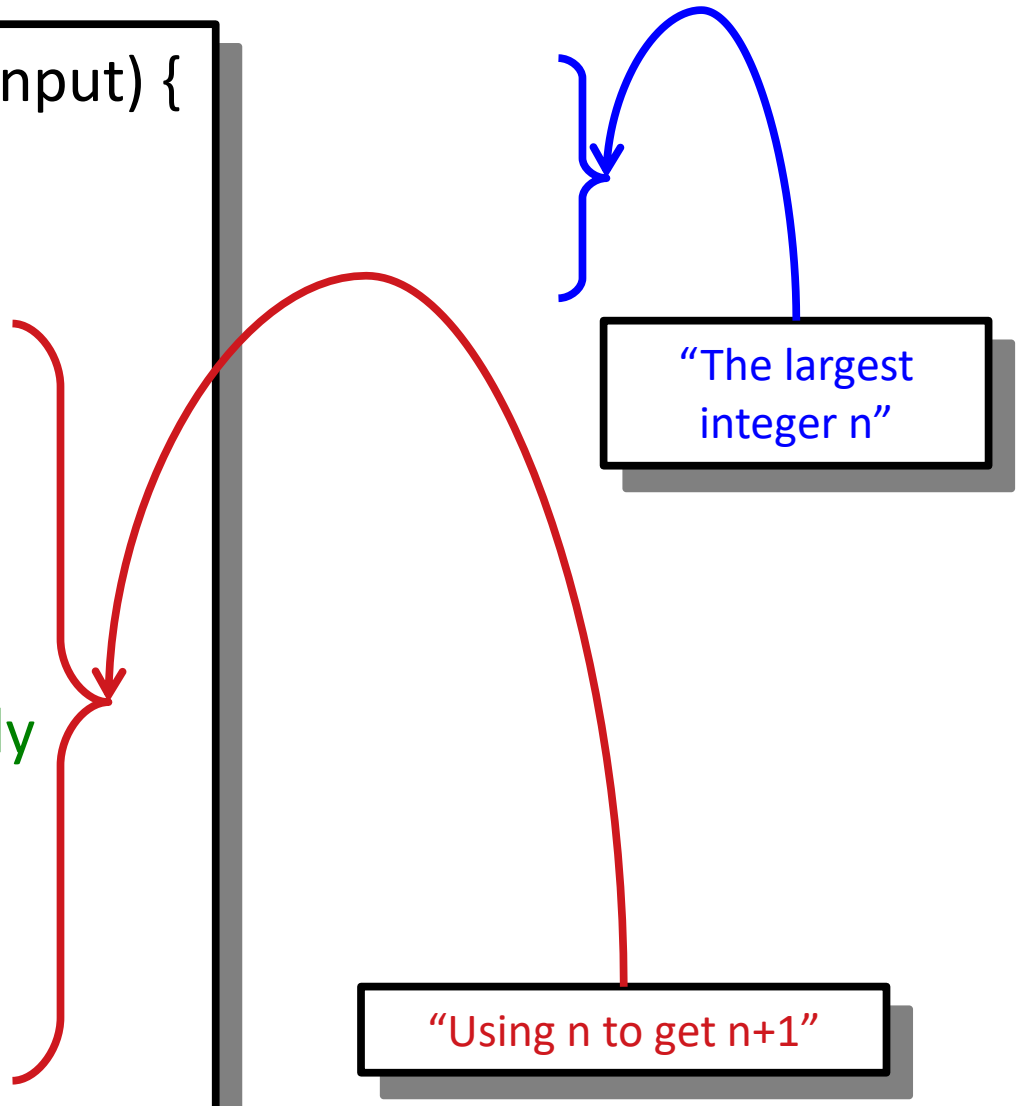
Imagine running this program on some input. What happens if...

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

… this program loops on this input?
It halts on 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();
        }
}
```



Theorem: $HALT \notin \mathbf{R}$.

Proof: By contradiction; assume that $HALT \in \mathbf{R}$. Then there's a decider D for $HALT$, which we can represent in software as a method willHalt that takes as input the source code of a program and an input, then returns true if the program halts on the input and false otherwise.

Given this, we could then construct this program P:

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

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

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the willHalt method. If willHalt(me, input) returns true, then P must halt on its input w. However, in this case P proceeds to loop infinitely on w. Otherwise, if willHalt(me, input) returns false, then P must not halt its input w. However, in this case P proceeds to accept its input w.

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

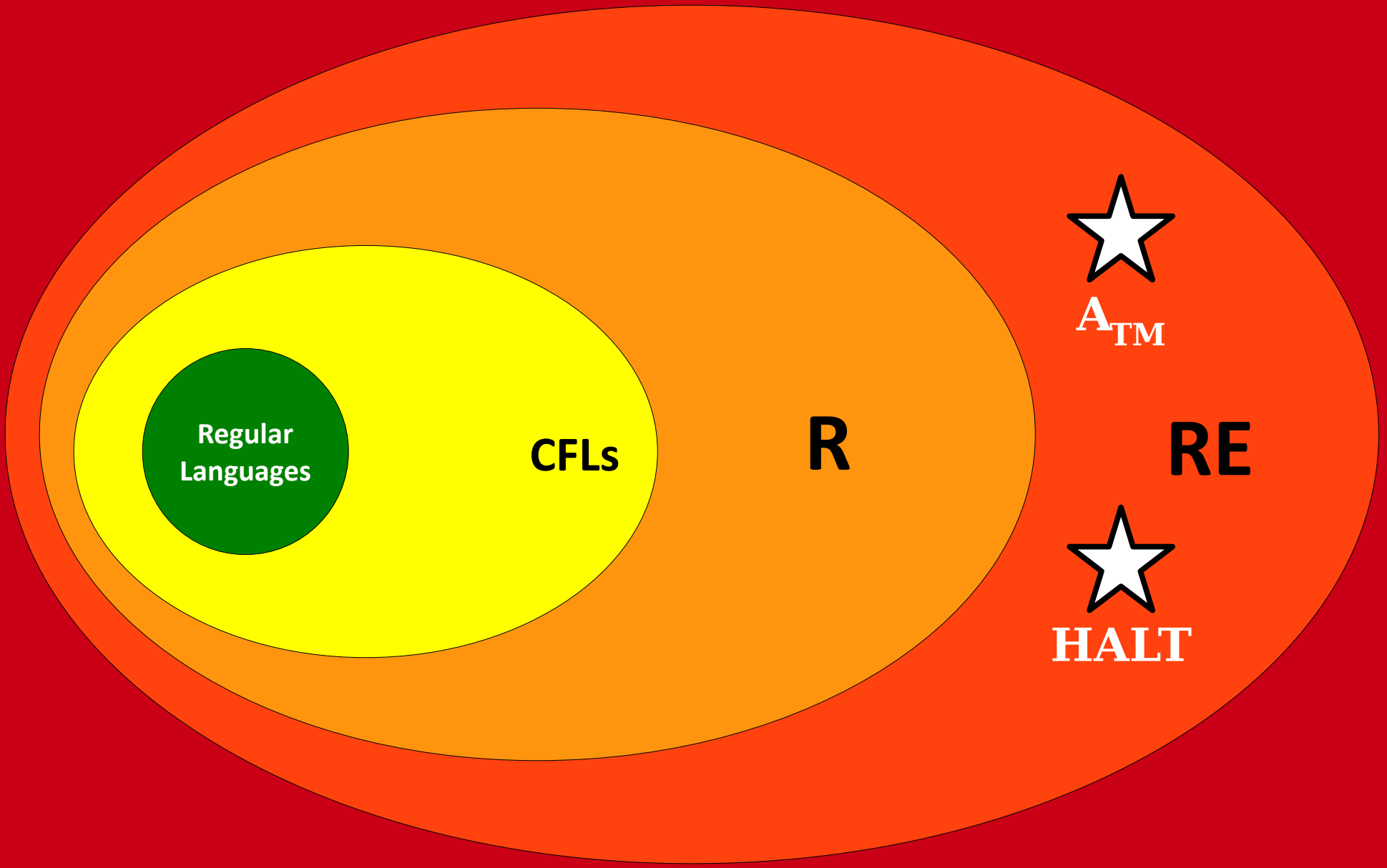
$HALT \in \mathbf{RE}$

Claim: $HALT \in \mathbf{RE}$.

Idea: If you were certain that a TM M halted on a string w, could you convince me of that?

Yes – just run M on w and see what happens!

```
int main() {
        TM M = getInputTM();
        string w = getInputString();
        feed w into M;
        while (true) {
                if (M is in an accepting state) accept();
                else if (M is in a rejecting state) accept();
                else simulate one more step of M running on w;
        }
}
```


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

So What?

These problems might not seem all that exciting, so who cares if we can't solve them?

Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.

Secure Voting

- Suppose that you want to make a voting machine for use in an election between two parties.
- Let $\Sigma = \{$**r**, **d**$\}$. A string in w corresponds to a series of votes for the candidates.
- Example: **rrdddrd** means “two people voted for **r**, then three people voted for **d**, then one more person voted for **r**, then one more person voted for **d**.”

Secure Voting

- A voting machine is a program that takes as input a string of **r**'s and **d**'s, then reports whether person **r** won the election.
- ***Question:*** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?

A secure voting machine is a TM M where
$\mathscr{L}(M) = \{ w \in \Sigma^* \mid w$ has more r's than d's $\}$

```
int main() {
  string input = getInput();
  int numRs = countRsIn(input);
  int numDs = countDsIn(input);

  if (numRs > numDs) accept();
  else reject();
}
```

A (simple) secure voting machine.

```
int main() {
  string input = getInput();

  if (input[0] == 'r') accept();
  else reject();
}
```

A (simple) insecure voting machine.

```
int main() {
  string input = getInput();
  int numRs = countRsIn(input);
  int numDs = countDsIn(input);

  if (numRs = numDs) reject();
  else if (numRs < numDs) reject();
  else accept();
}
```

An (evil) insecure voting machine.

```
int main() {
  string input = getInput();
  int n = input.length();
  while (n > 1) {
    if (n % 2 == 0) n /= 2;
    else n = 3*n + 1;
  }
  int numRs = countRsIn(input);
  int numDs = countDsIn(input);

  if (numRs > numDs) accept();
  else reject();
}
```

No one knows!

Secure Voting

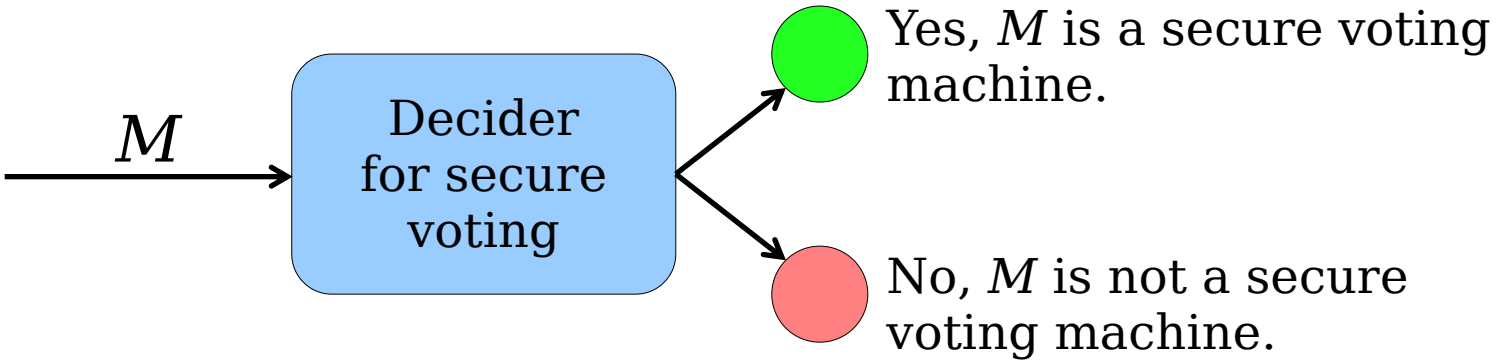
- A voting machine is a program that takes as input a string of **r**'s and **d**'s, then reports whether person **r** won the election.
- ***Question:*** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?

Claim: A program that decides whether arbitrary input programs are secure voting machines is self-defeating. It therefore doesn’t exist.

A Decider for Secure Voting

Let's suppose that, somehow, we managed to build a decider for the secure voting problem.

Schematically, that decider would look like this:



We could represent this decider in software as a method

```
bool isSecureVotingMachine(string program);
```

that takes as input a program, then returns whether that program is a secure voting machine.

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

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;

  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;

  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?
then it's not a secure voting machine!

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?
then it's not a secure voting machine!

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?
then it's not 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();

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?
then it's not 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();

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if…

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?
then it's a secure voting machine!

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

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

  bool answer = countRs(input) > countDs(input);
        if (isSecureVotingMachine(me)) answer = !answer;


  if (answer) accept();
  else reject();
}
```

What happens if...

... this program is a secure voting machine?
then it's not a secure voting machine!

... this program is not a secure voting machine?
then it's a secure voting machine!

Theorem: The secure voting problem is undecidable.

Proof: By contradiction; assume that the secure voting problem is decidable. Then there is some decider D for the secure voting problem, which we can represent in software as a method isSecureVotingMachine that, given as input the source code of a program, returns true if the program is a secure voting machine and false otherwise.

Given this, we could then construct the following program P:

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

    bool answer = (countRs(input) > countDs(input));
        if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

Now, either P is a secure voting machine or it isn't. If P is a secure voting machine, then isSecureVotingMachine(me) will return true. Therefore, when P is run, it will determine whether w has more **r**'s than **d**'s, flip the result, and accept strings with at least as many **d**'s as **r**'s and reject strings with more **r**'s than **d**'s. Thus, P is not a secure voting machine. On the other hand, if P is not a secure voting machine, then isSecureVotingMachine(me) will return false. Therefore, when P is run, it will accept all strings with at least as many **r**'s as **d**'s and reject all other strings, and so P is a secure voting machine.

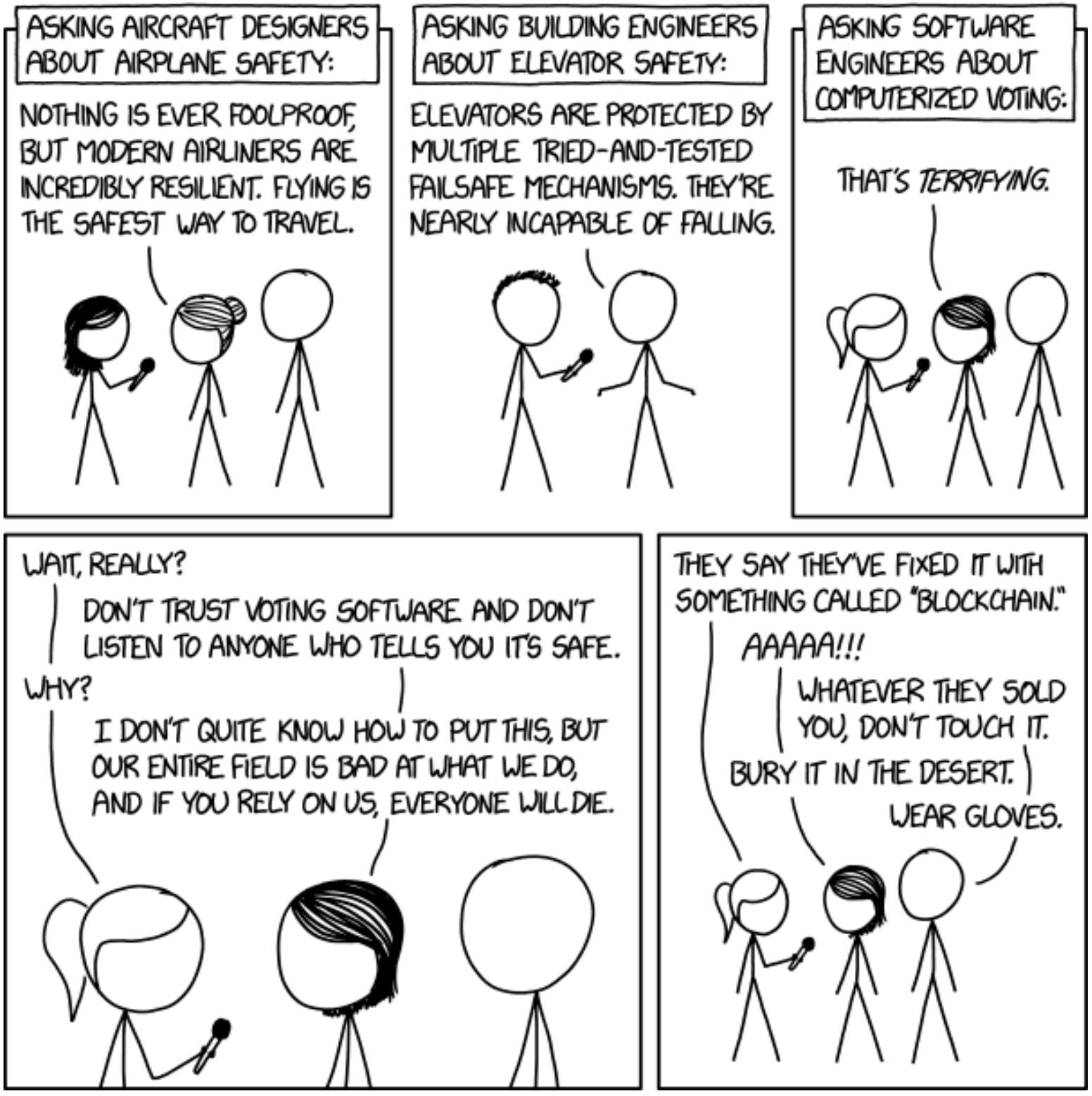
In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, the secure voting problem is undecidable. ■

Interpreting this Result

The previous argument tells us that *there is no general algorithm* that we can follow to determine whether a program is a secure voting machine. In other words, any general algorithm to check voting machines will always be wrong on at least one input.

So what can we do?

- Design algorithms that work in *some*, but not *all* cases. (This is often done in practice.)
- Fall back on human verification of voting machines. (We do that too.)
- Carry a healthy degree of skepticism about electronic voting machines. (Then again, did we even need the theoretical result for this?)


ASKING AIRCRAFT DESIGNERS ABOUT AIRPLANE SAFETY:
NOTHING IS EVER FOOLPROOF, BUT MODERN AIRLINERS ARE INCREDIBLY RESILIENT. FLYING IS THE SAFEST WAY TO TRAVEL.
ASKING BUILDING ENGINEERS ABOUT ELEVATOR SAFETY:
ELEVATORS ARE PROTECTED BY MULTIPLE TRIED-AND-TESTED FAILSAFE MECHANISMS. THEY'RE NEARLY INCAPABLE OF FALLING.
ASKING SOFTWARE ENGINEERS ABOUT COMPUTERIZED VOTING:
THAT'S TERRIFYING.
WAIT, REALLY?
DON'T TRUST VOTING SOFTWARE. AND DON'T LISTEN TO ANYONE WHO TELLS YOU IT'S SAFE.
WHY?
I DON'T QUITE KNOW HOW TO PUT THIS, BUT OUR ENTIRE FIELD IS BAD AT WHAT WE DO, AND IF YOU RELY ON US, EVERYONE WILL DIE.
THEY SAY THEY'VE FIXED IT WITH SOMETHING CALLED "BLOCKCHAIN."
AAAAA!!!
WHATEVER THEY SOLD YOU, DON'T TOUCH IT.
BURY IT IN THE DESERT.
WEAR GLOVES.

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

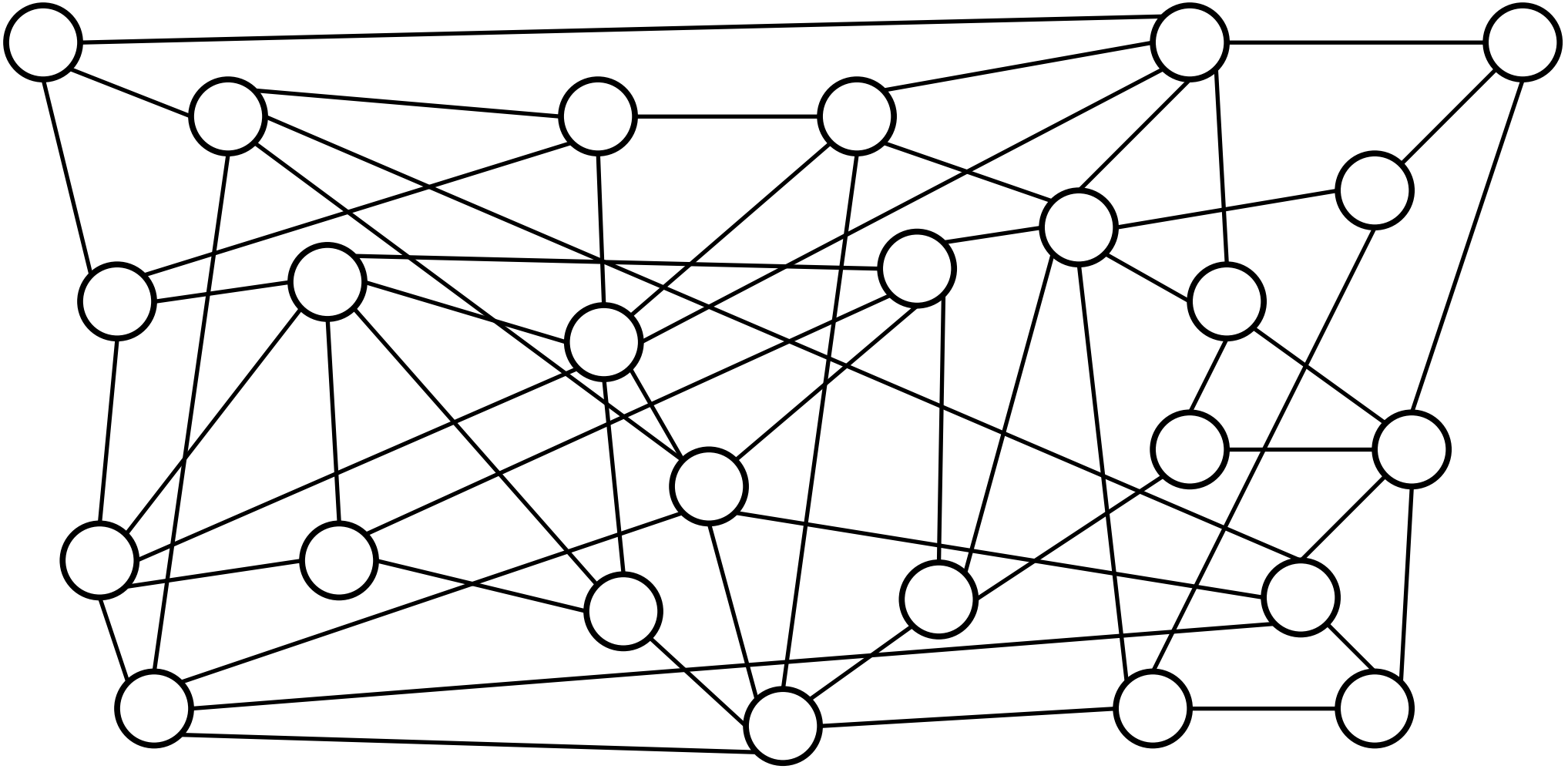
Beyond **R** and **RE**

- 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**.

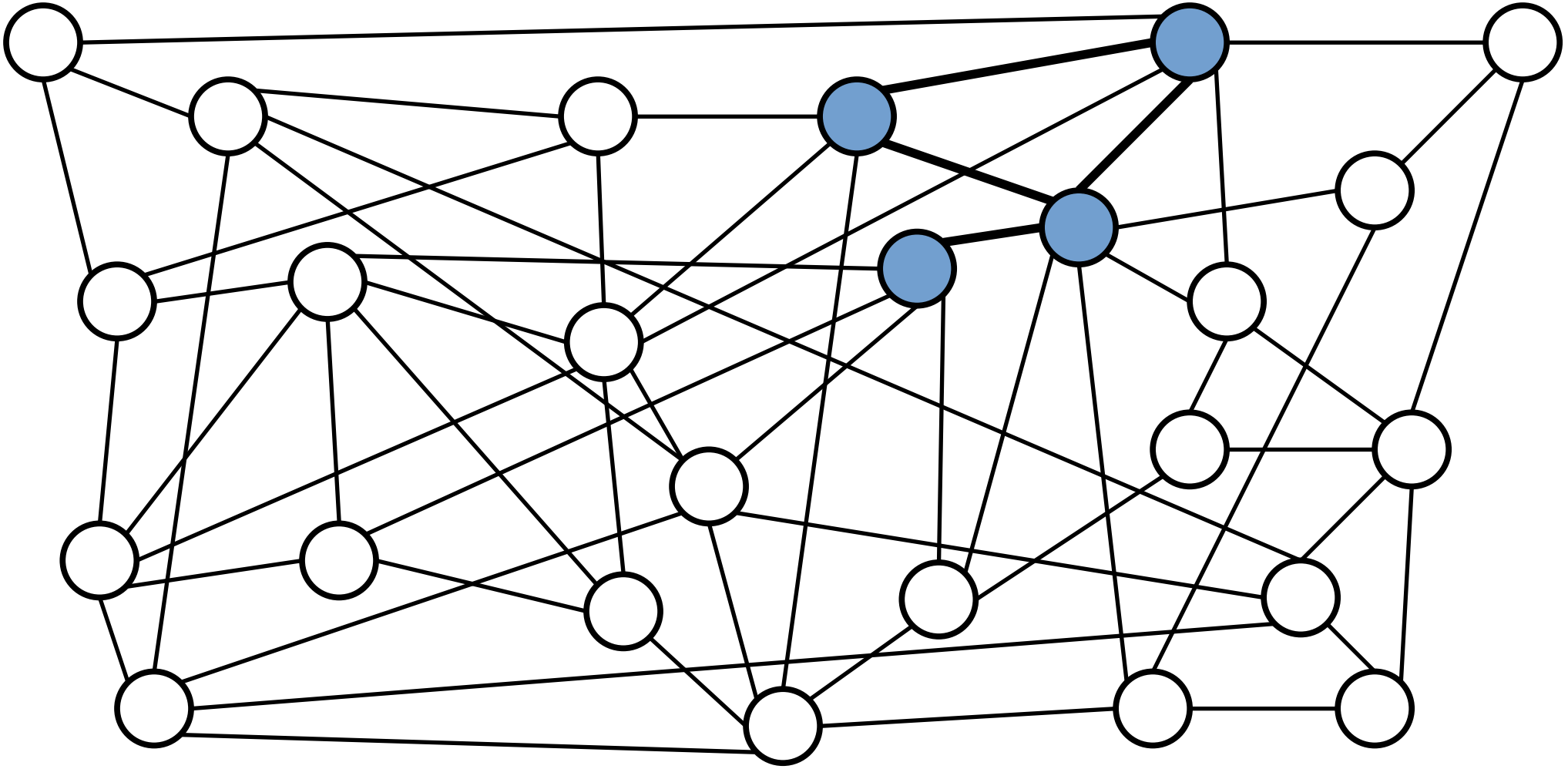
What exactly is the class **RE**?

RE, Formally

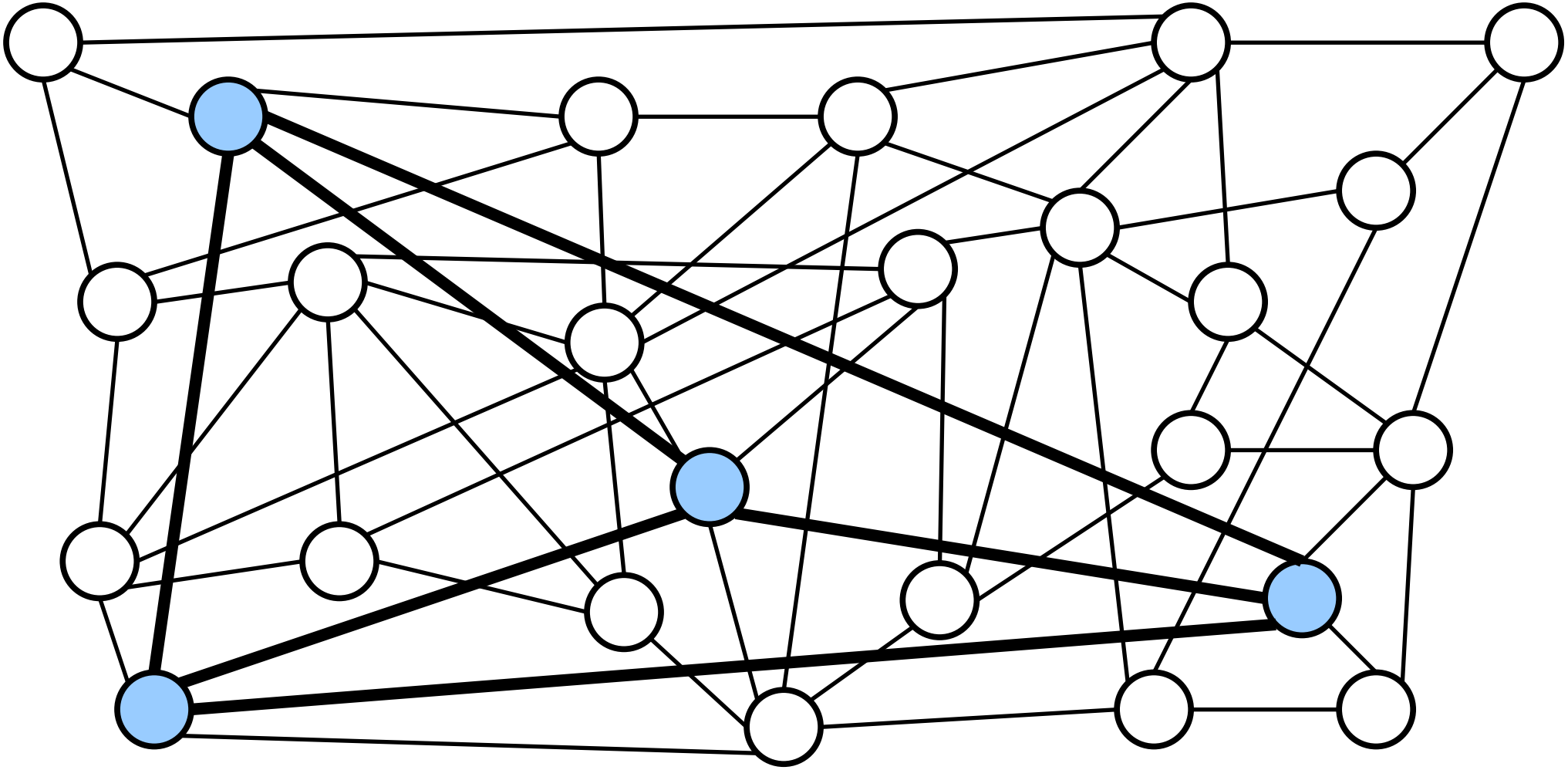
- Recall that the class **RE** is the class of all recognizable languages:
  - **RE** = { L | there is a TM M where $\mathscr{L}(M) = L$ }
- Since **R** ≠ **RE**, there is no general way to "solve" problems in the class **RE**, if by "solve" you mean "make a computer program that can always tell you the correct answer."
- So what exactly *are* the sorts of languages in **RE**?



Does this graph contain a 4-clique?



Does this graph contain a 4-clique?



Does this graph contain a 4-clique?

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

		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		

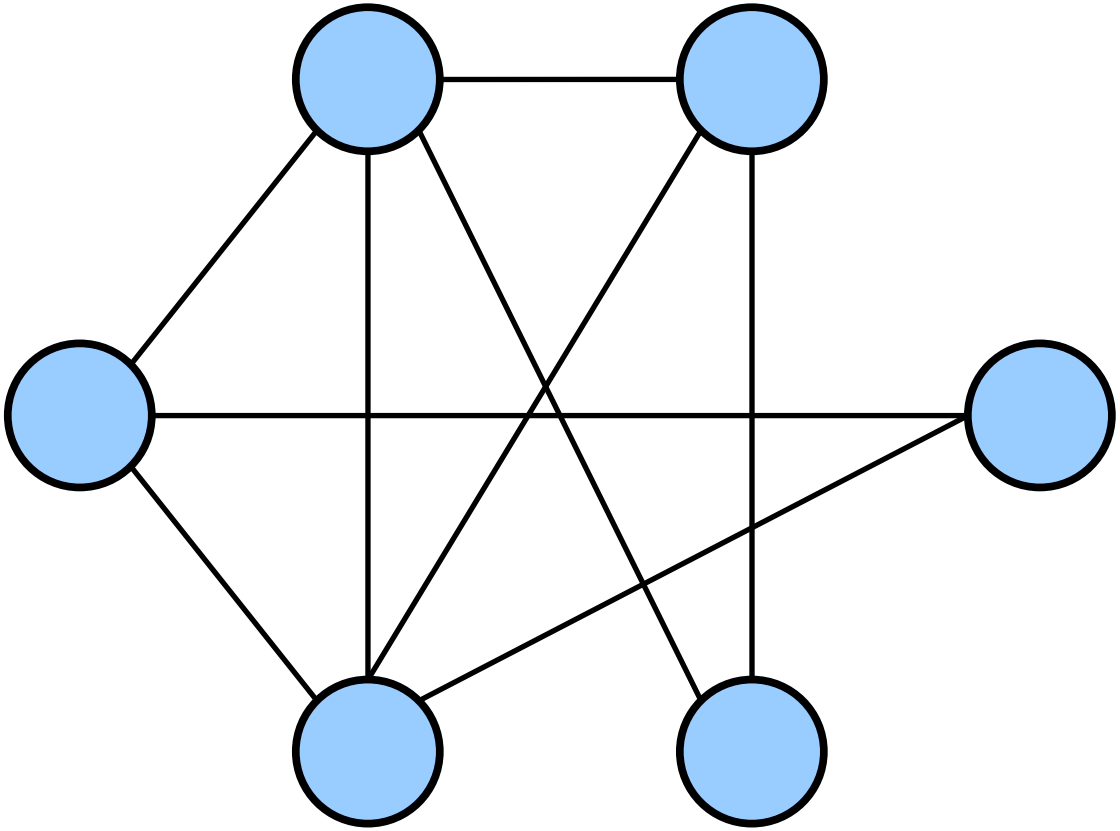
Does this Sudoku puzzle have a solution?

Verification

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

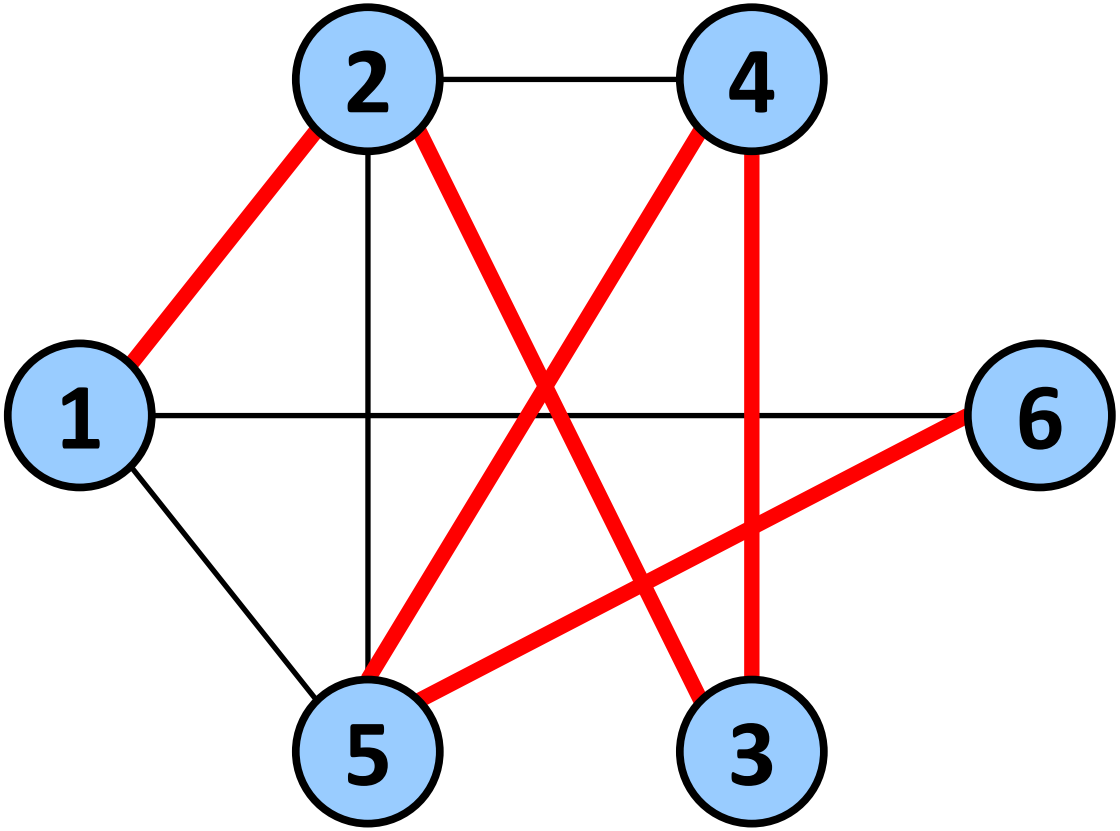
Does this Sudoku puzzle have a solution?

Verification



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

Verification



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

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?

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		

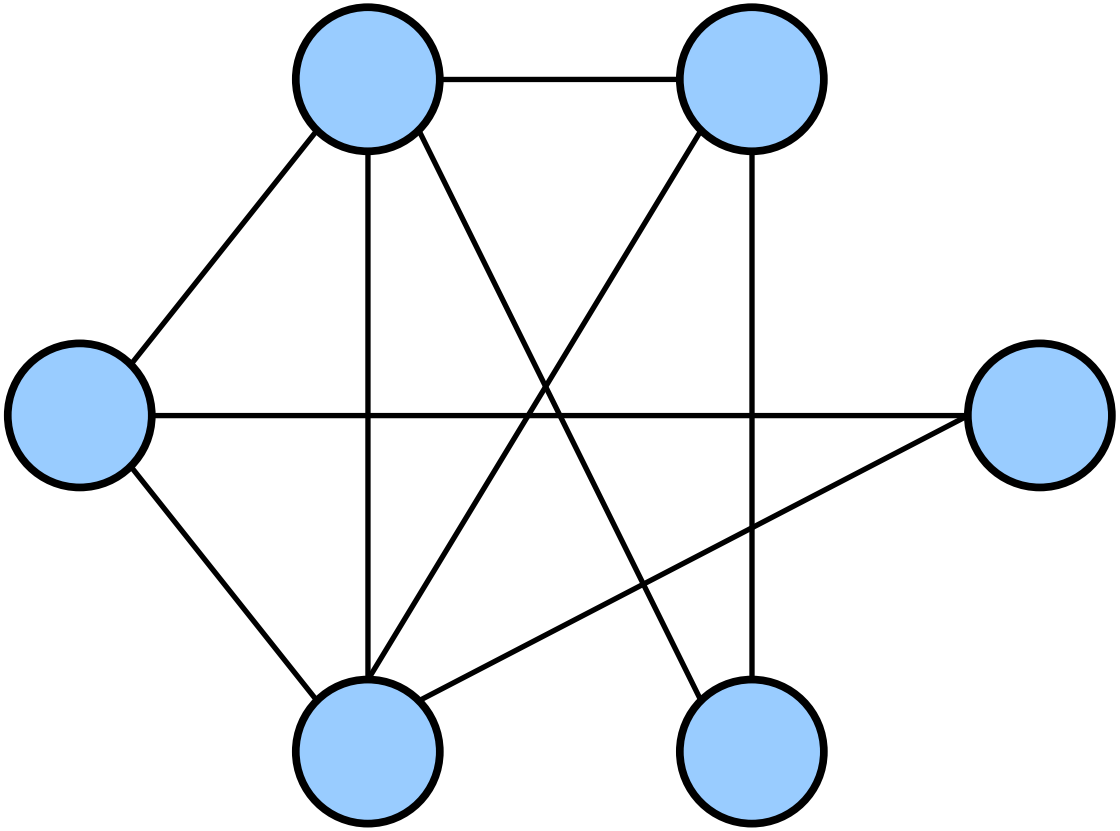
Does this Sudoku puzzle have a solution?

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

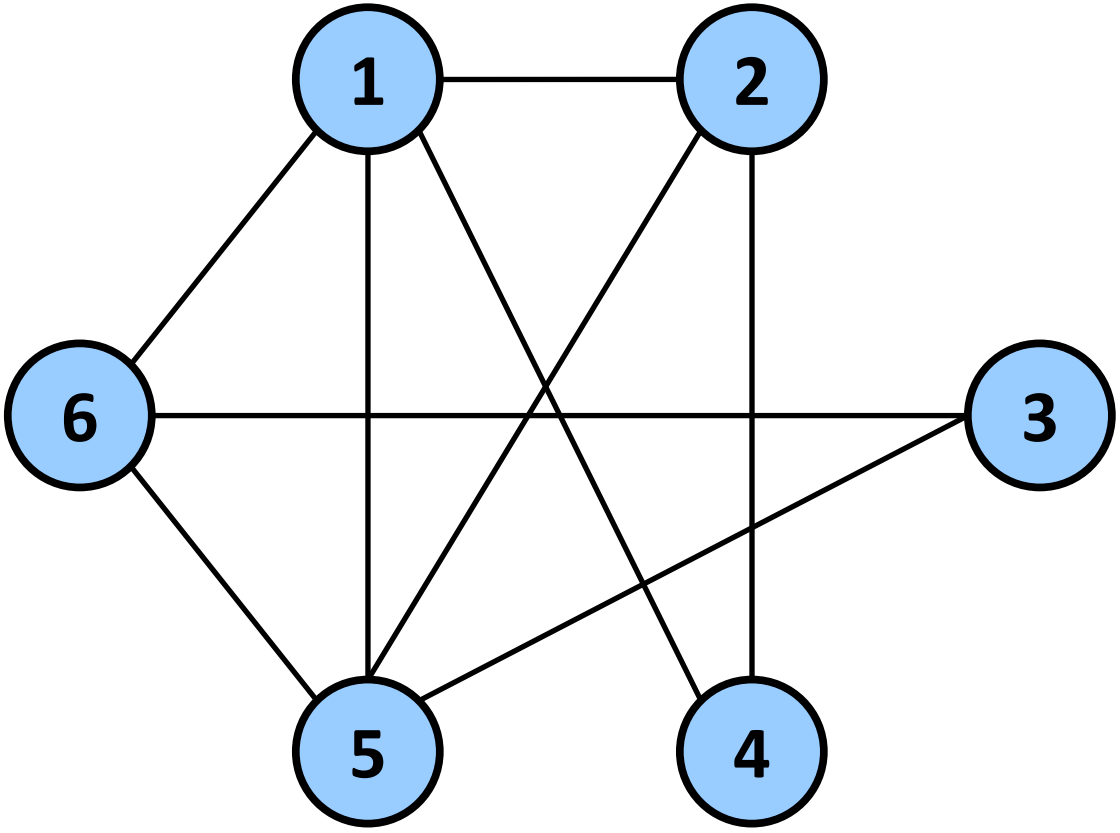
Does this Sudoku puzzle have a solution?

Verification



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

Verification



Does this graph have a ***Hamiltonian path*** (a simple path that passes 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

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

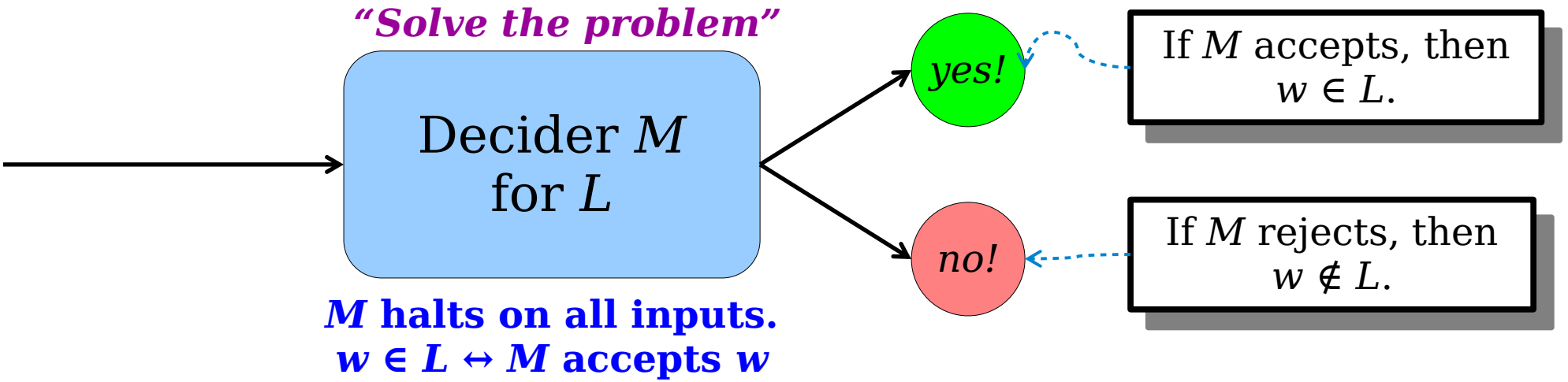
A ***verifier*** for a language L is a TM V with the following two properties:

V halts on all inputs.

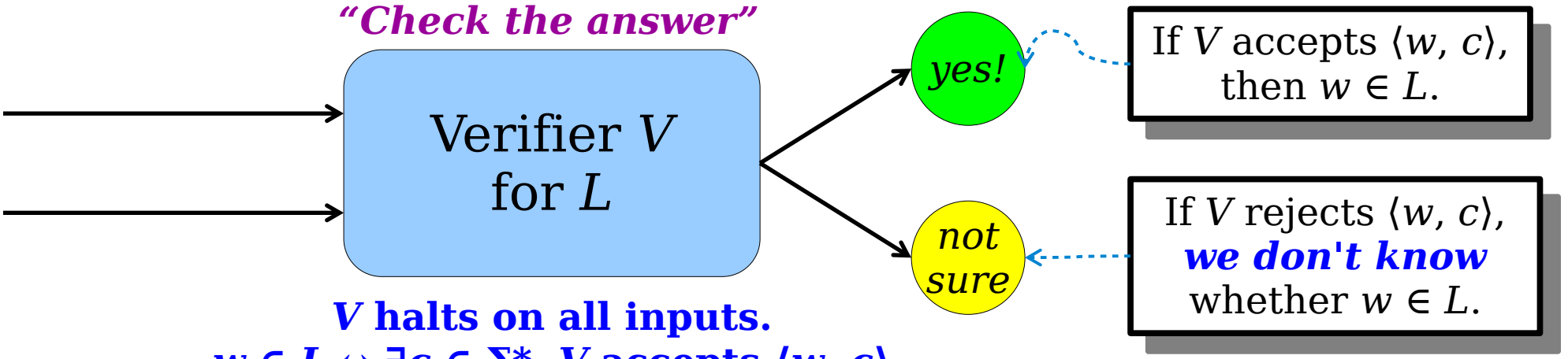
$\forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*.$ V accepts $\langle w, c \rangle)$

Intuitively, what does this mean?

Deciders and Verifiers



M halts on all inputs.
$w \in L \leftrightarrow M$ accepts w



V halts on all inputs.
$w \in L \leftrightarrow \exists c \in \Sigma^*$. V accepts $\langle w, c \rangle$

Verifiers

A ***verifier*** for a language L is a TM V with the following properties:

V halts on all inputs.

$\forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

Some notes about V:

- If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
- If V rejects $\langle w, c \rangle$, 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.

$\forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

Some notes about V:

- Notice that the certificate 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.

$\forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

Some notes about V:

- Notice that $\mathscr{L}(V) \neq L$. *(Good question: what <u>is</u> $\mathscr{L}(V)$?)*
- 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.

$\forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

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.

This verifier will take as input

- a natural number n, and
- some certificate c.

The certificate c should be some evidence that suggests that the hailstone sequence terminates for n.

What evidence could we provide?

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?

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:

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

(Refresher: a Hamiltonian path is a simple path that visits every node in the graph.)

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

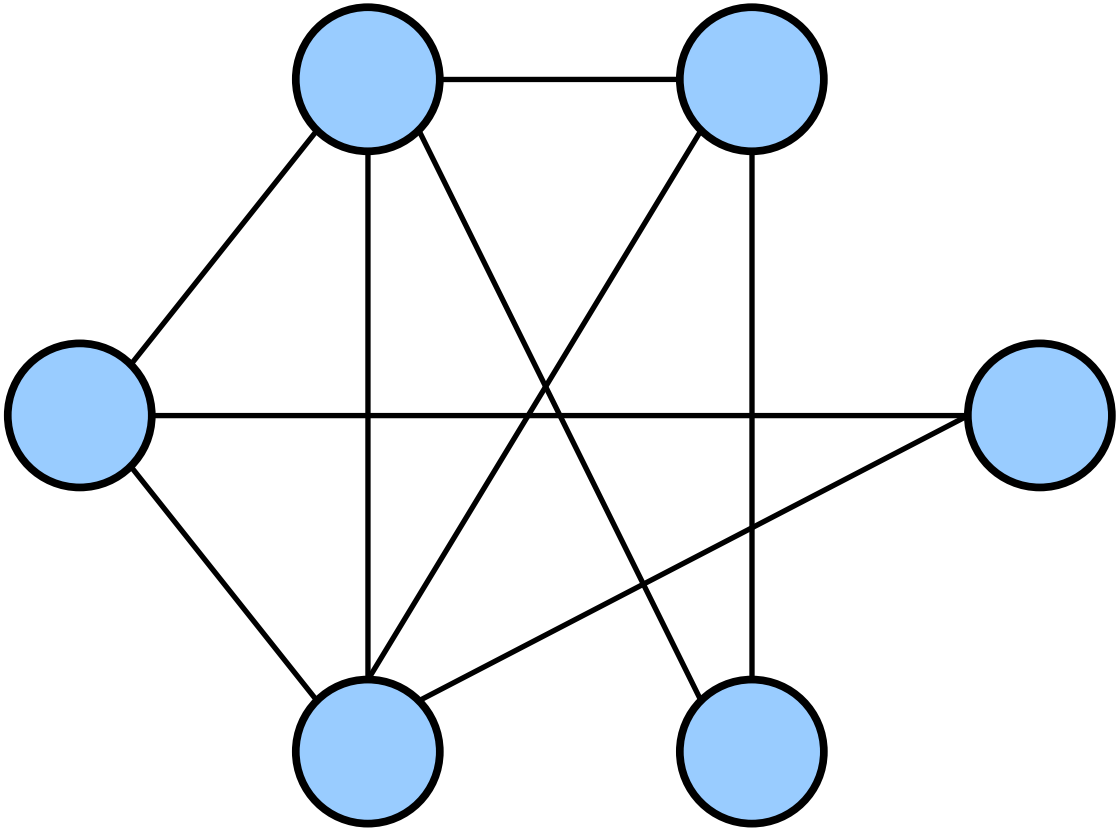
Our verifier will take as input

- a graph G, and
- a certificate c.

The certificate c should be some evidence that suggests that G has a Hamiltonian path.

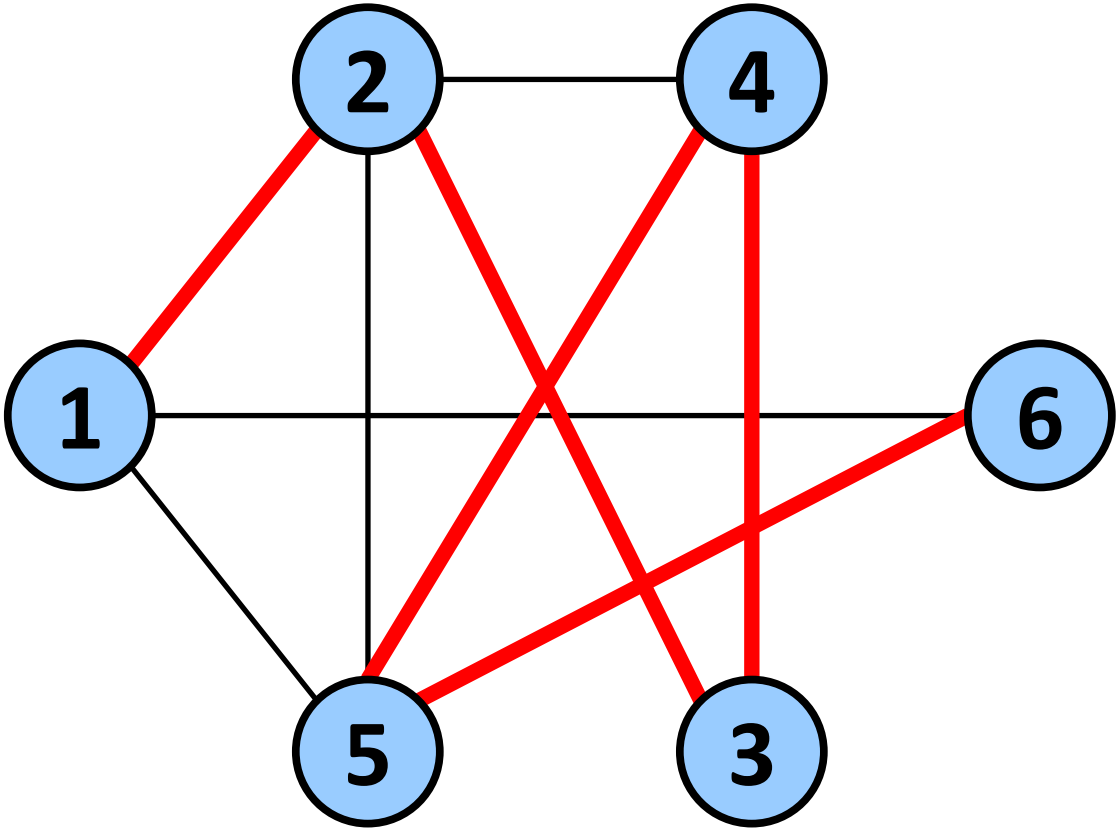
What information could we put into the certificate?

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?

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 + 1 < c.size(); 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?

A Very Nifty Verifier

Consider A_{TM}:

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

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!

A Very Nifty Verifier

Consider A_{TM}:

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

We know that U_{TM} is a recognizer for A_{TM}. It is also a *verifier* for A_{TM}?

No, for two reasons:

- U_{TM} doesn't always halt. *(Do you see why?)*
- U_{TM} takes as input a TM M and a string w. A verifier also needs a certificate.

A Very Nifty Verifier

Consider A_{TM}:

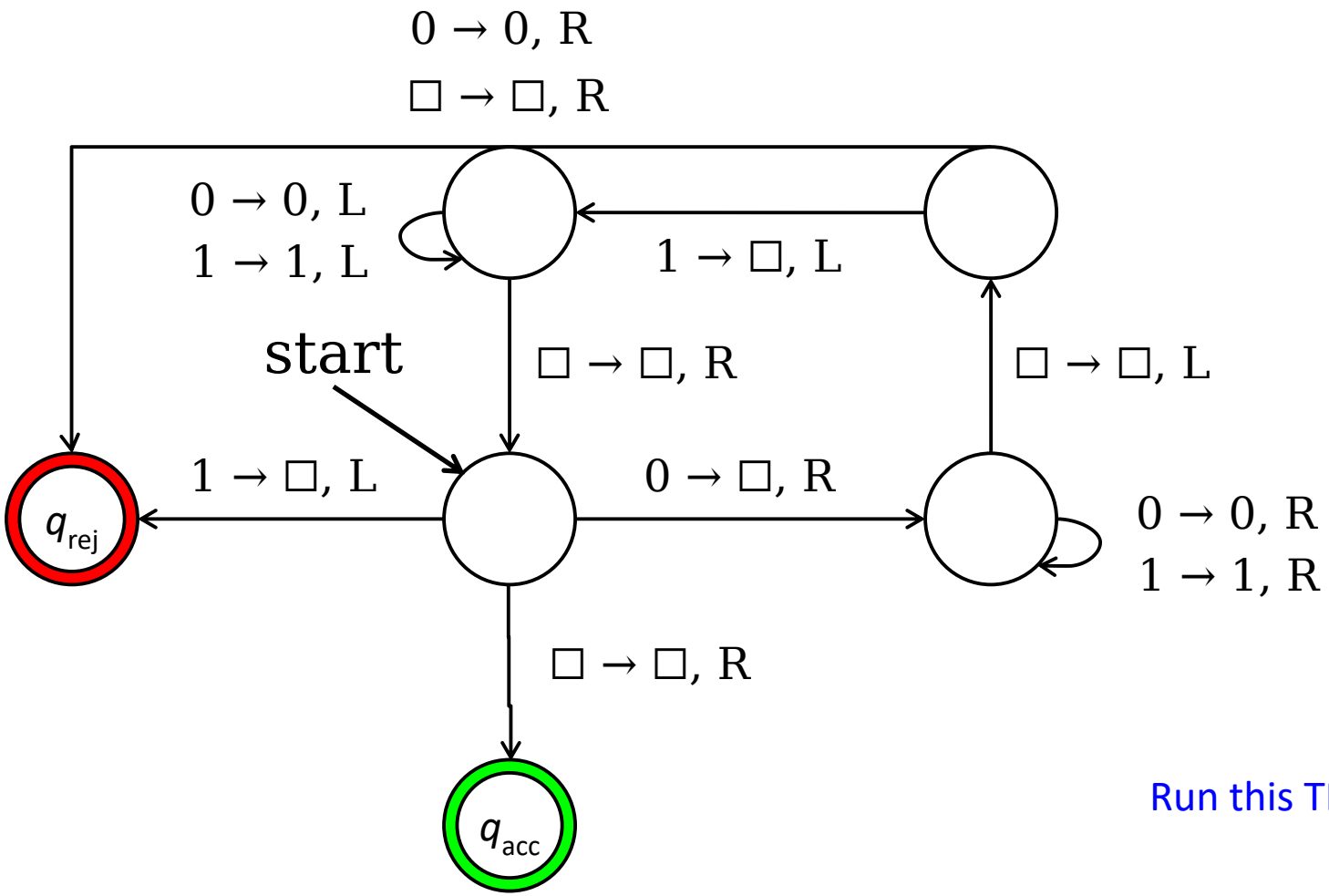
$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$.

A verifier for A_{TM} would take as input

- A TM M,
- a string w, and
- a certificate c.

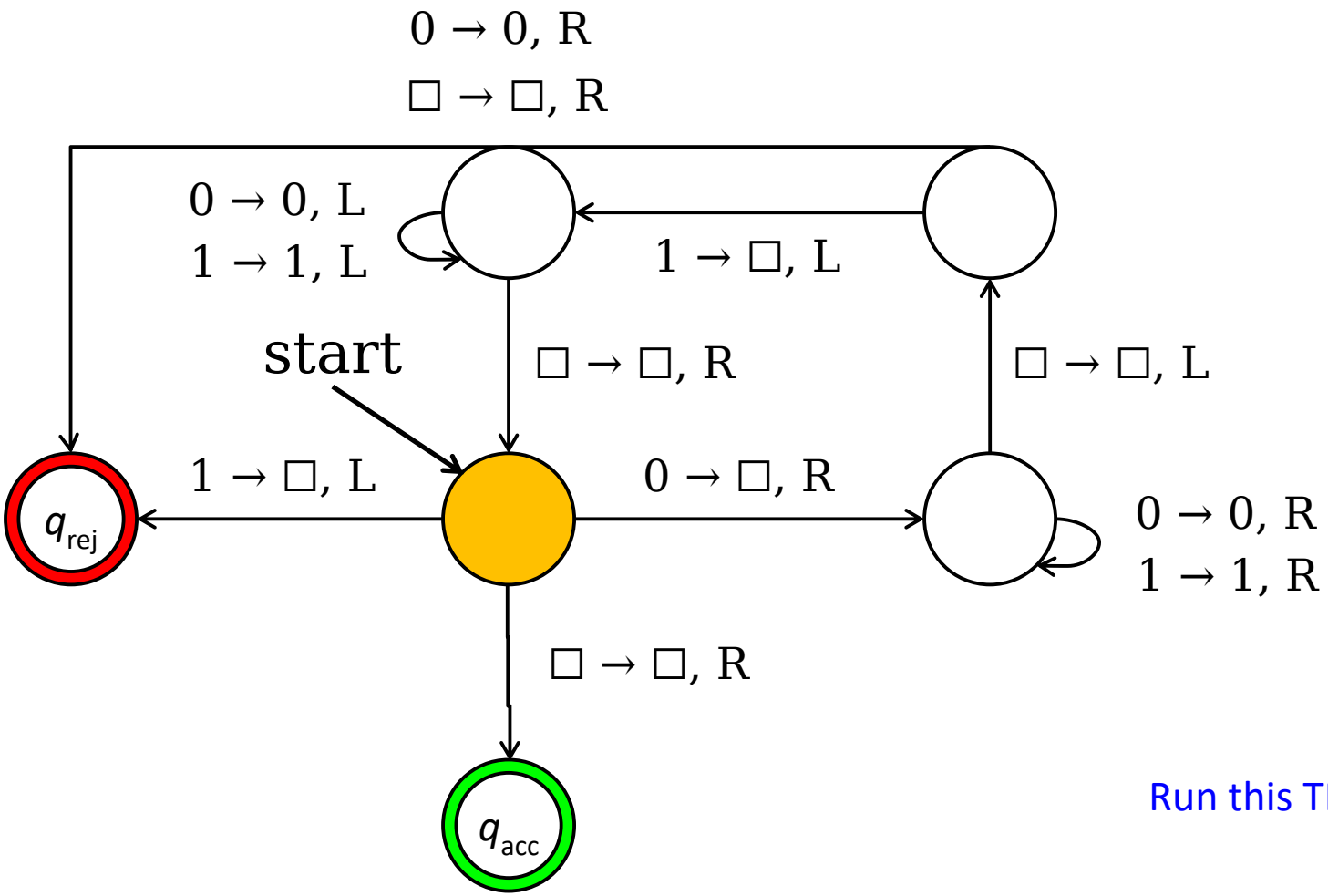
The certificate c should be some evidence that suggests that M accepts w.

What could our certificate be?

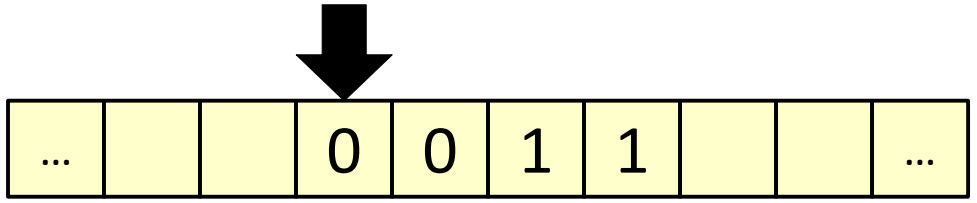


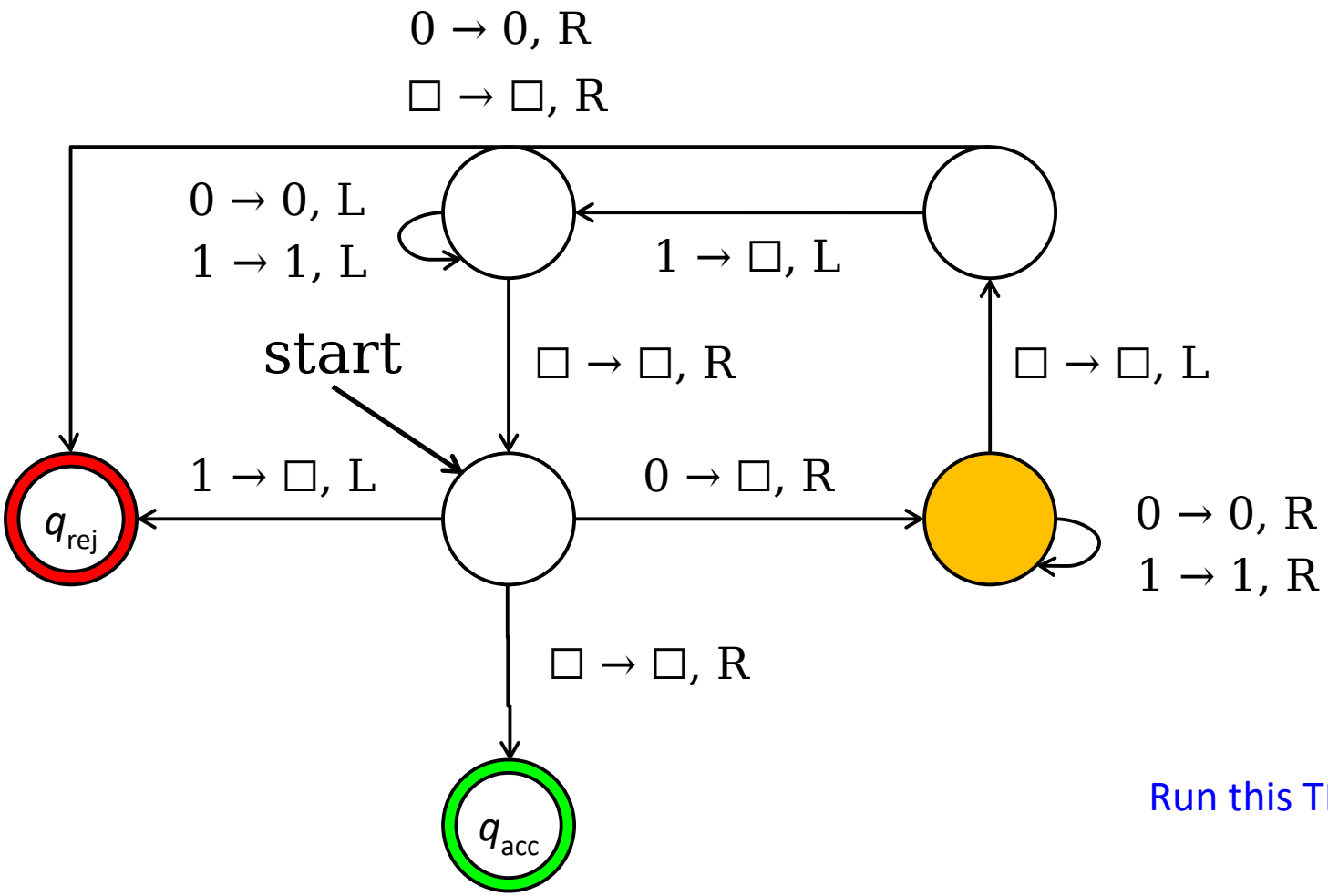
Run this TM for fifteen steps.

…			0	0	1	1			…

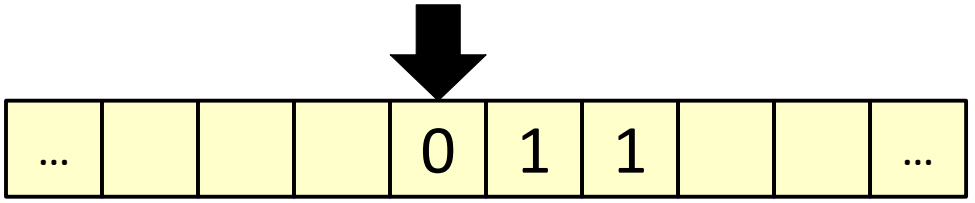


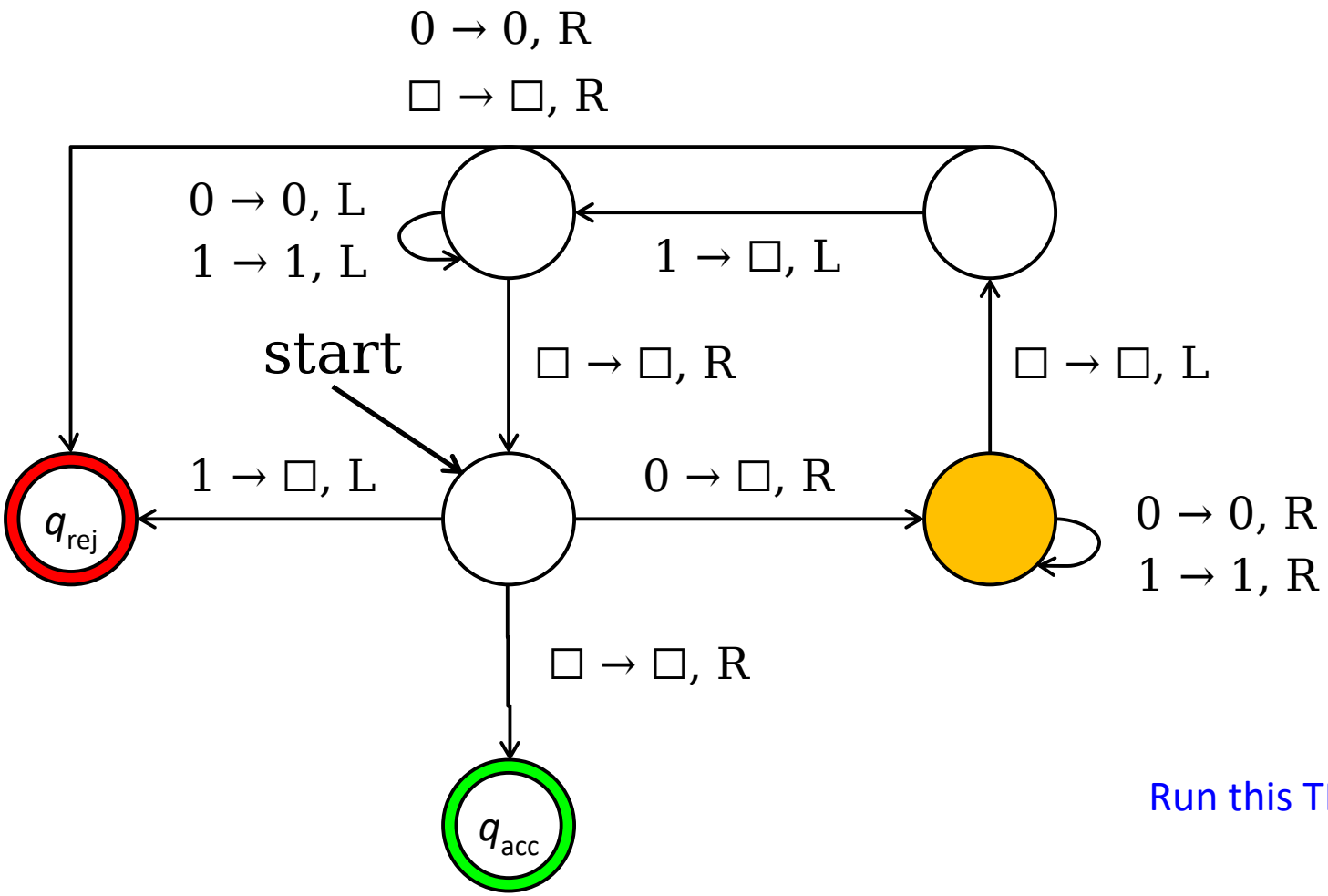
Run this TM for fifteen steps.



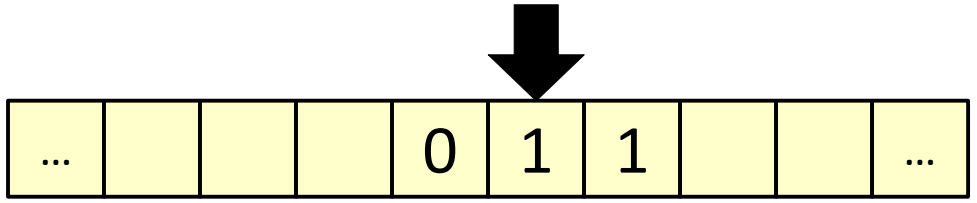


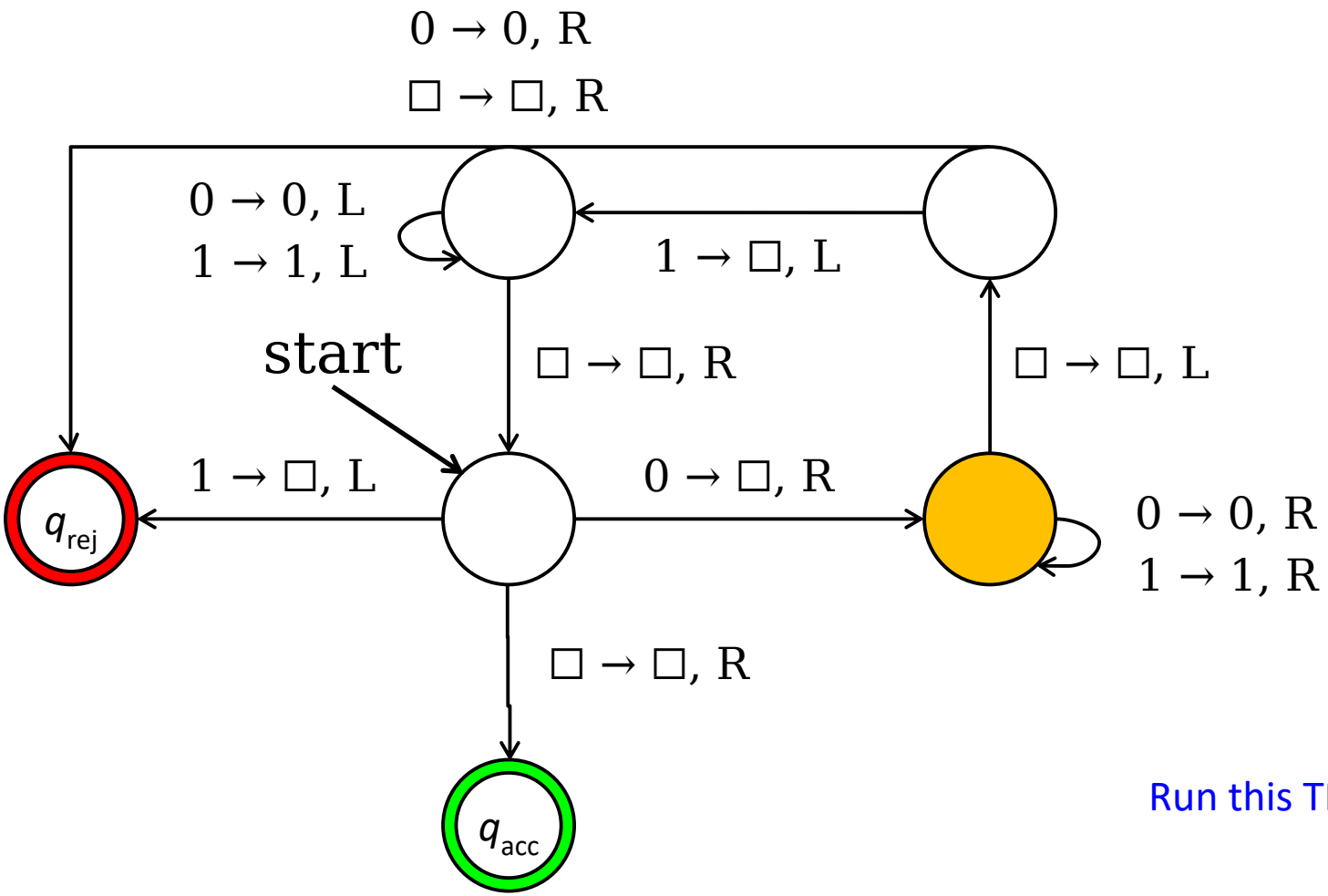
Run this TM for fifteen steps.



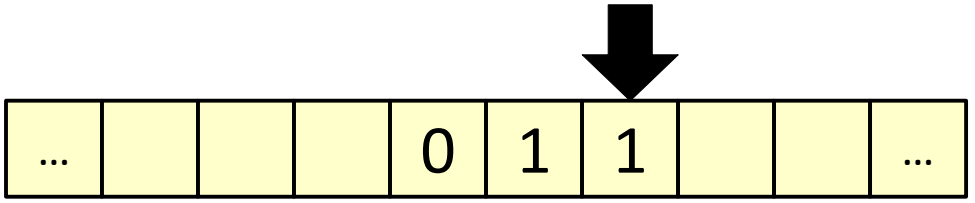


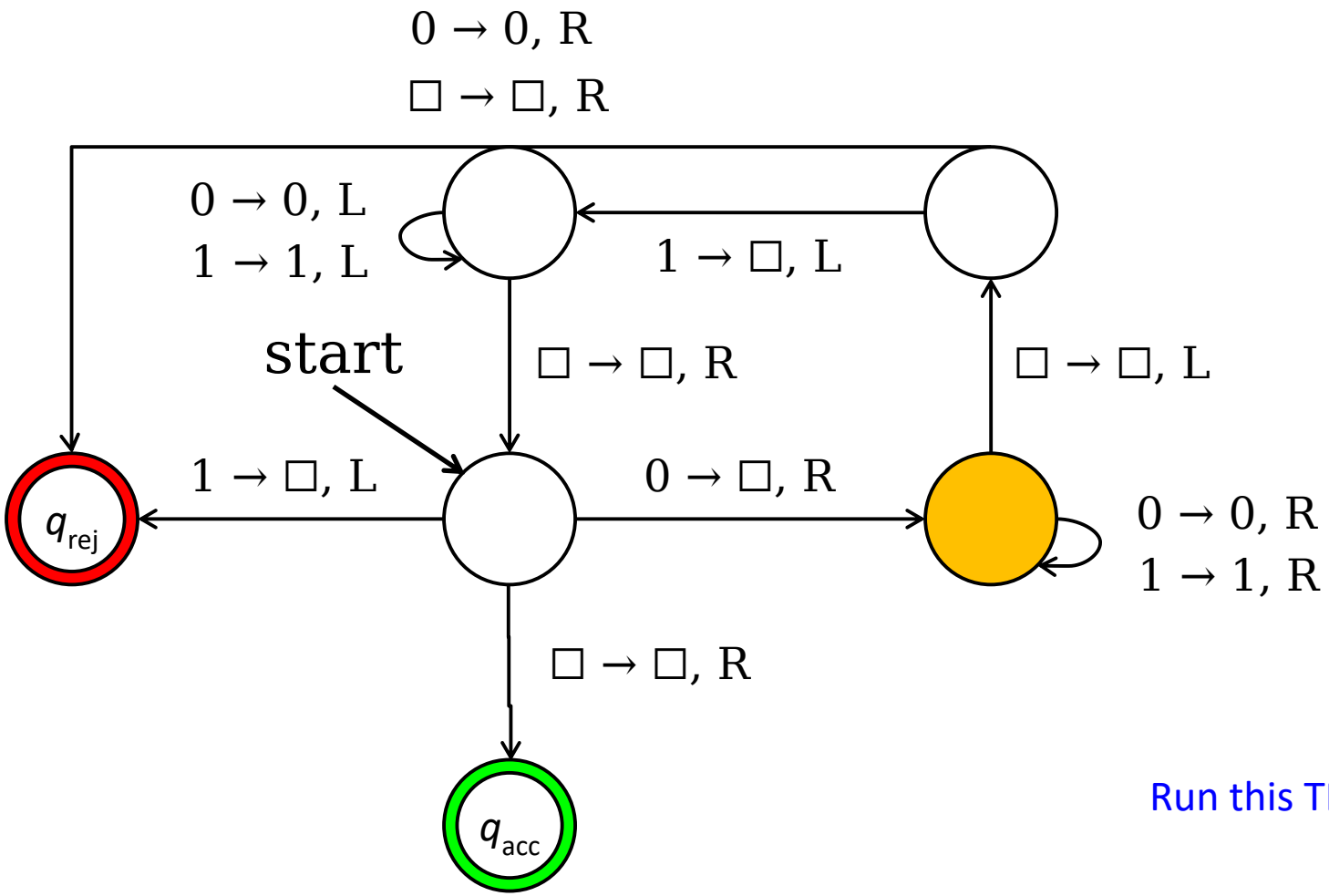
Run this TM for fifteen steps.



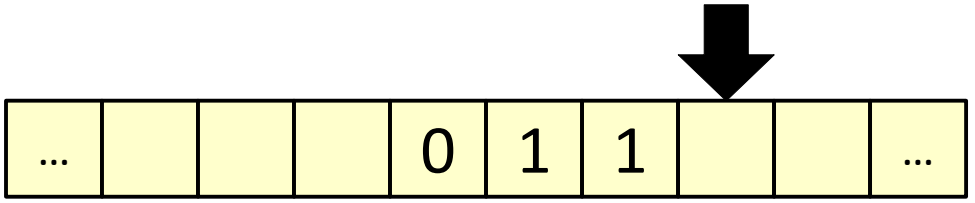


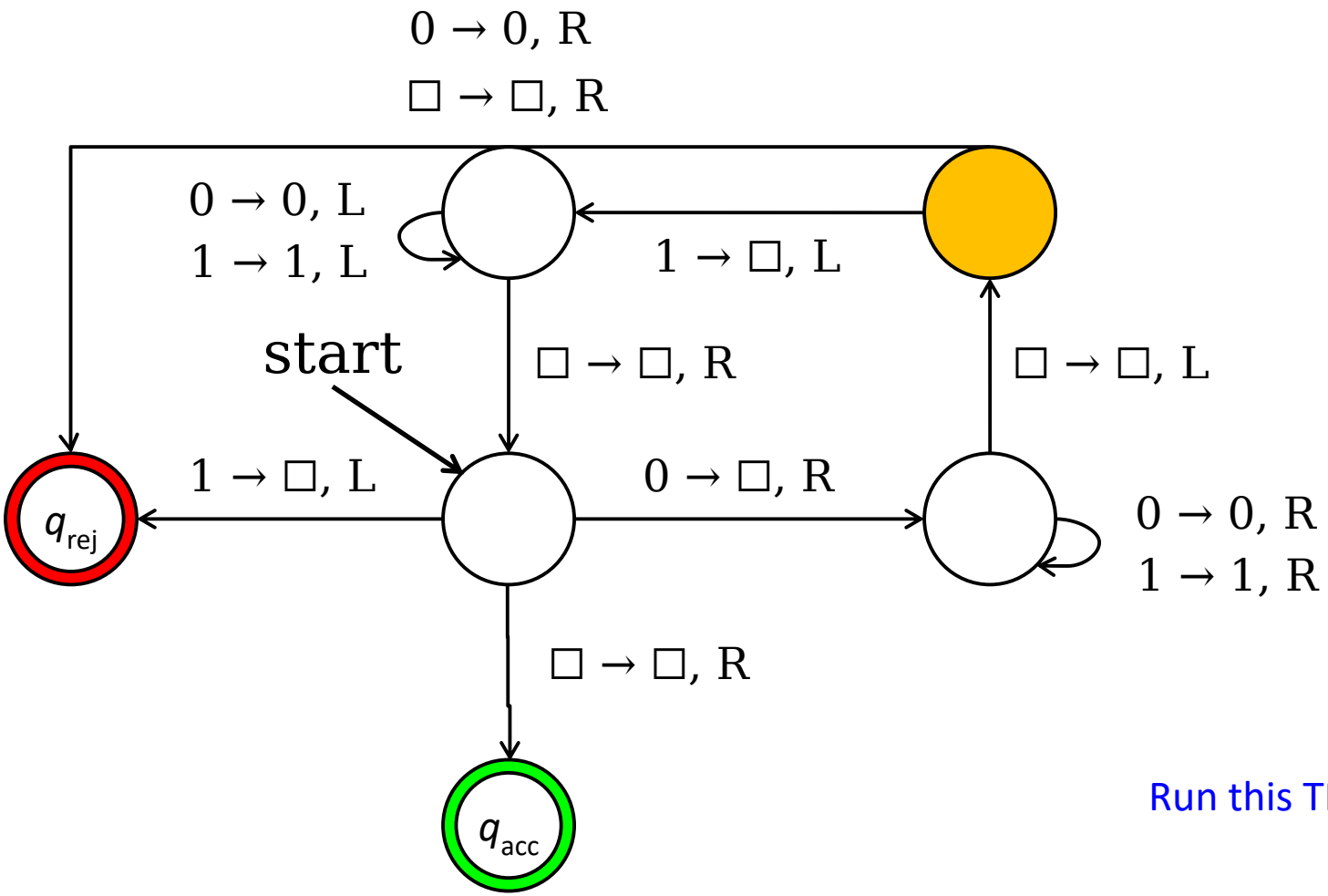
Run this TM for fifteen steps.



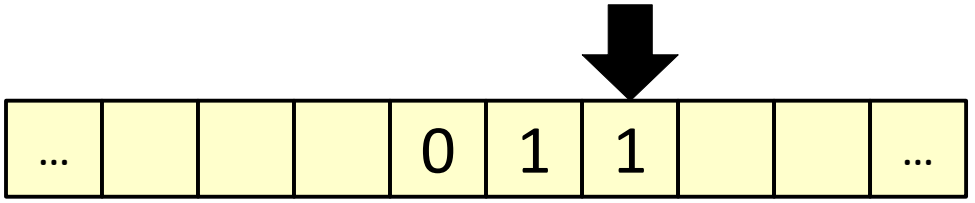


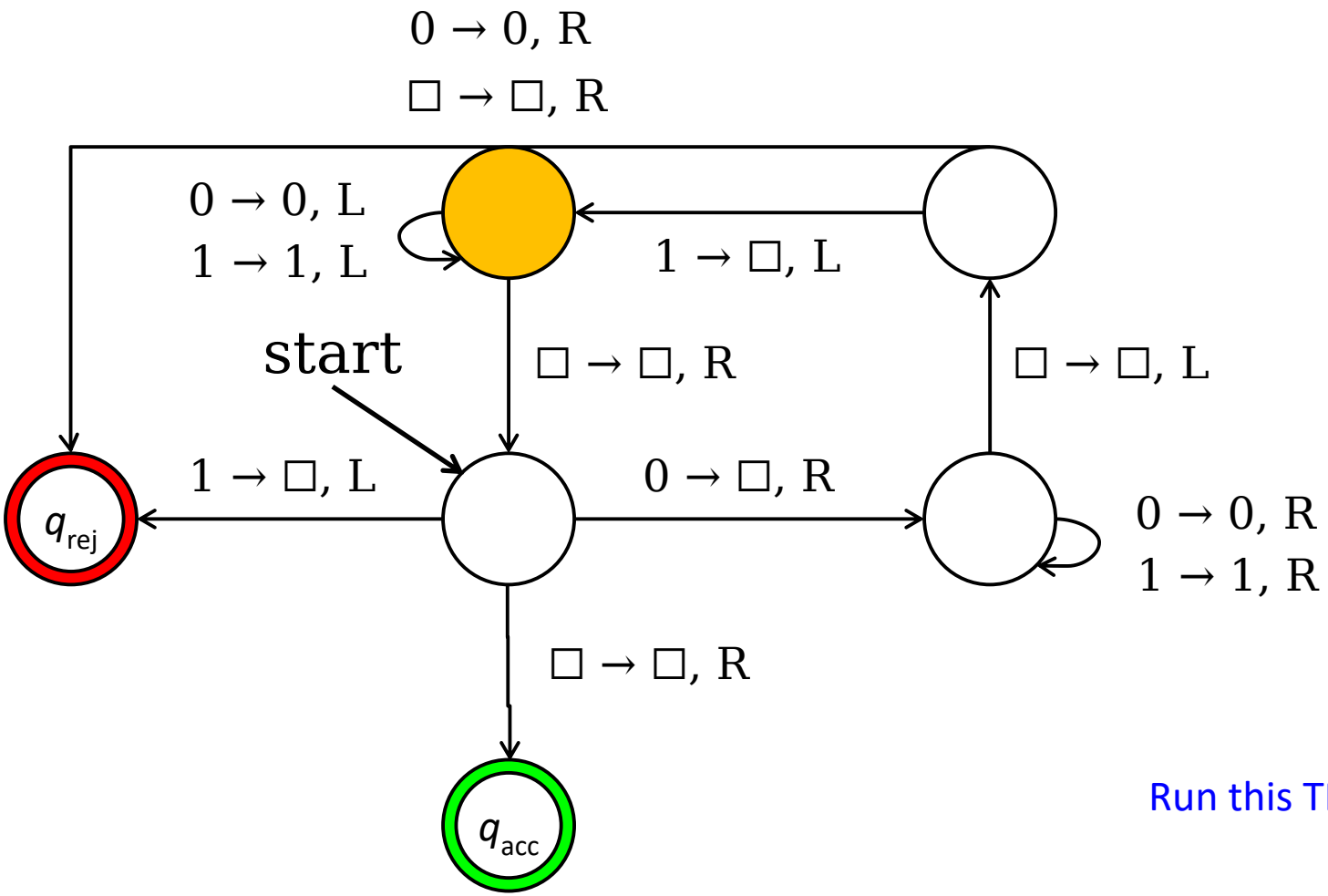
Run this TM for fifteen steps.



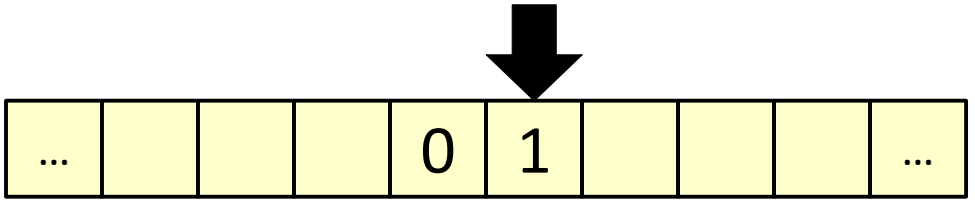


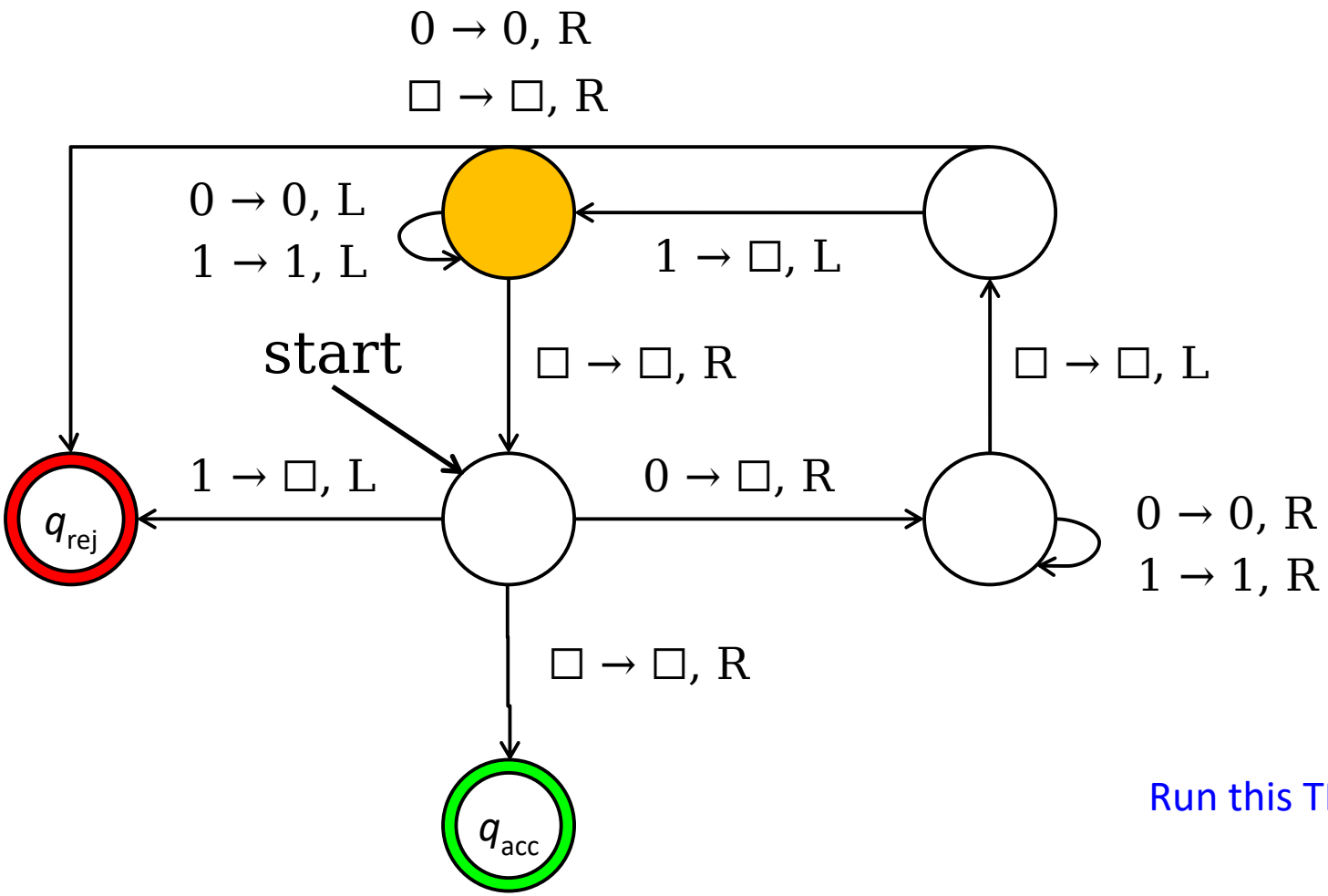
Run this TM for fifteen steps.



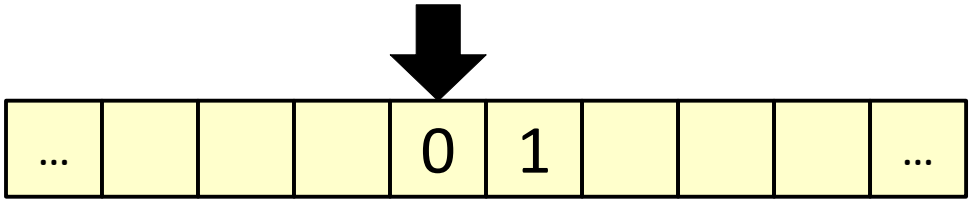


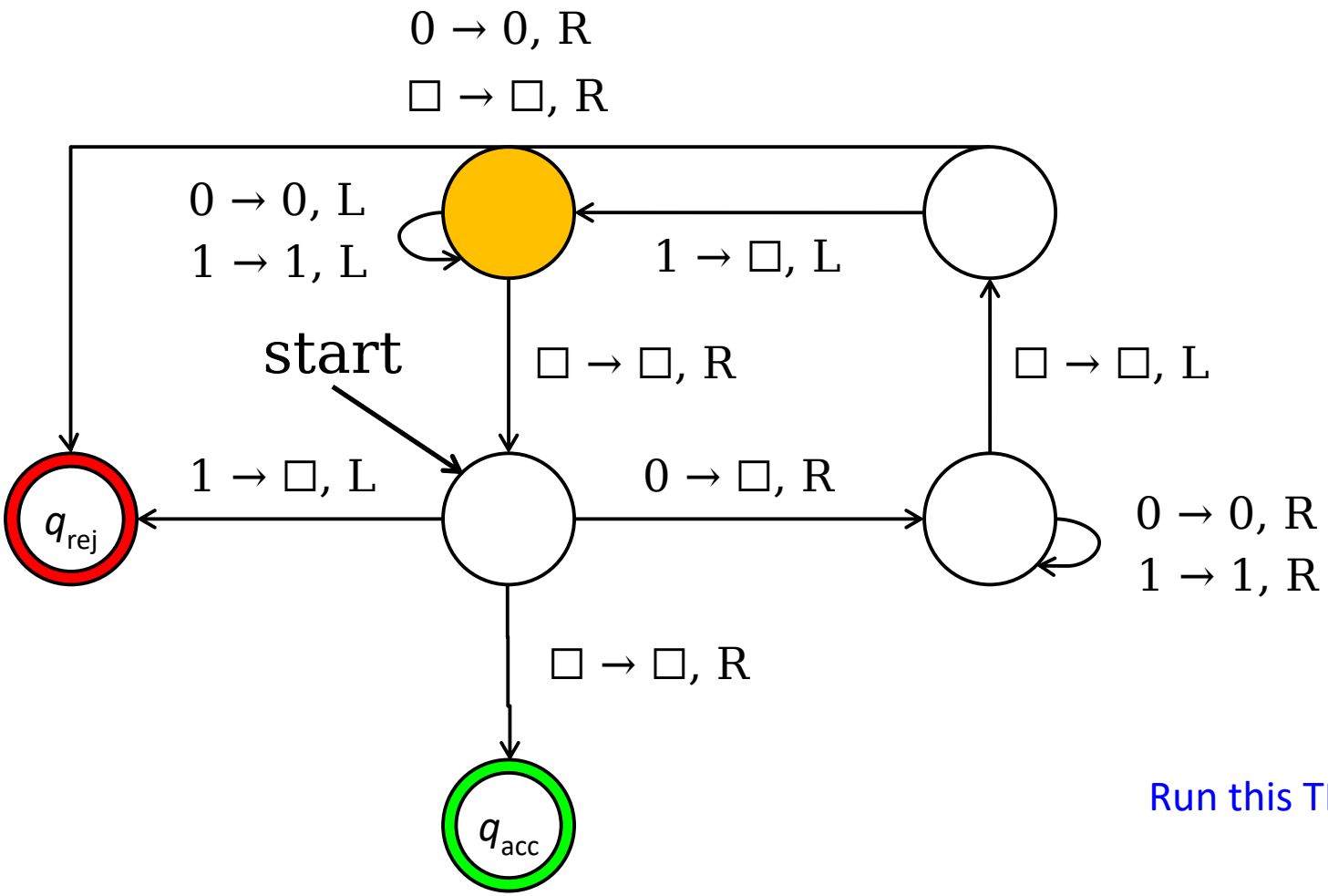
Run this TM for fifteen steps.



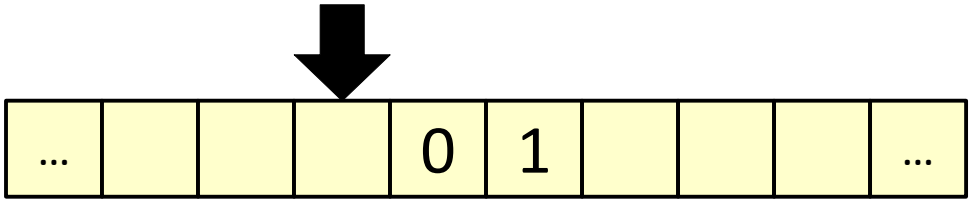


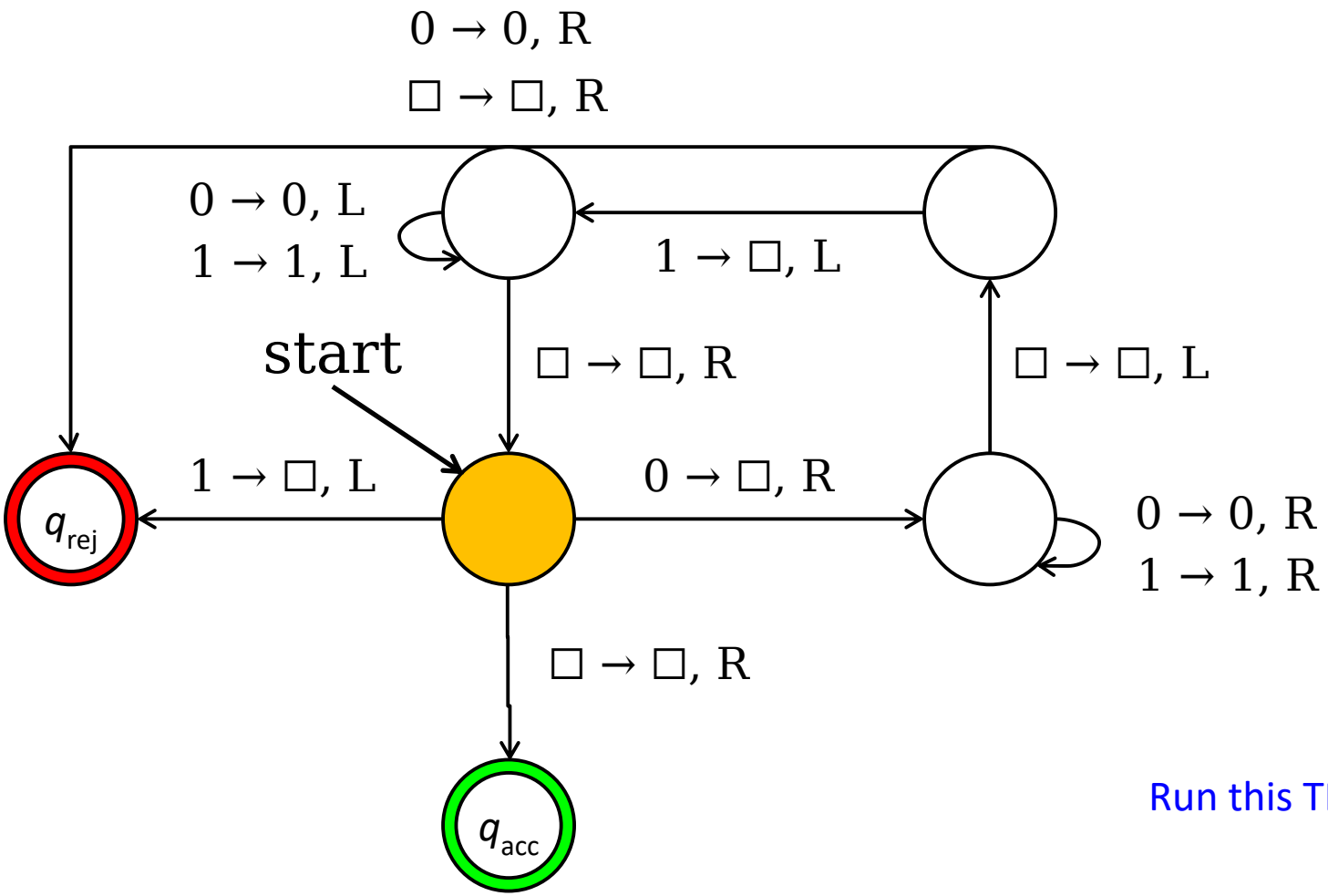
Run this TM for fifteen steps.



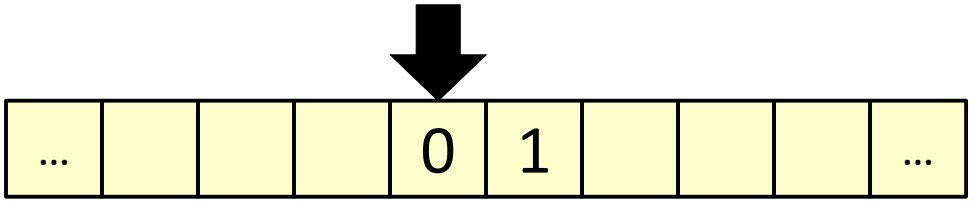


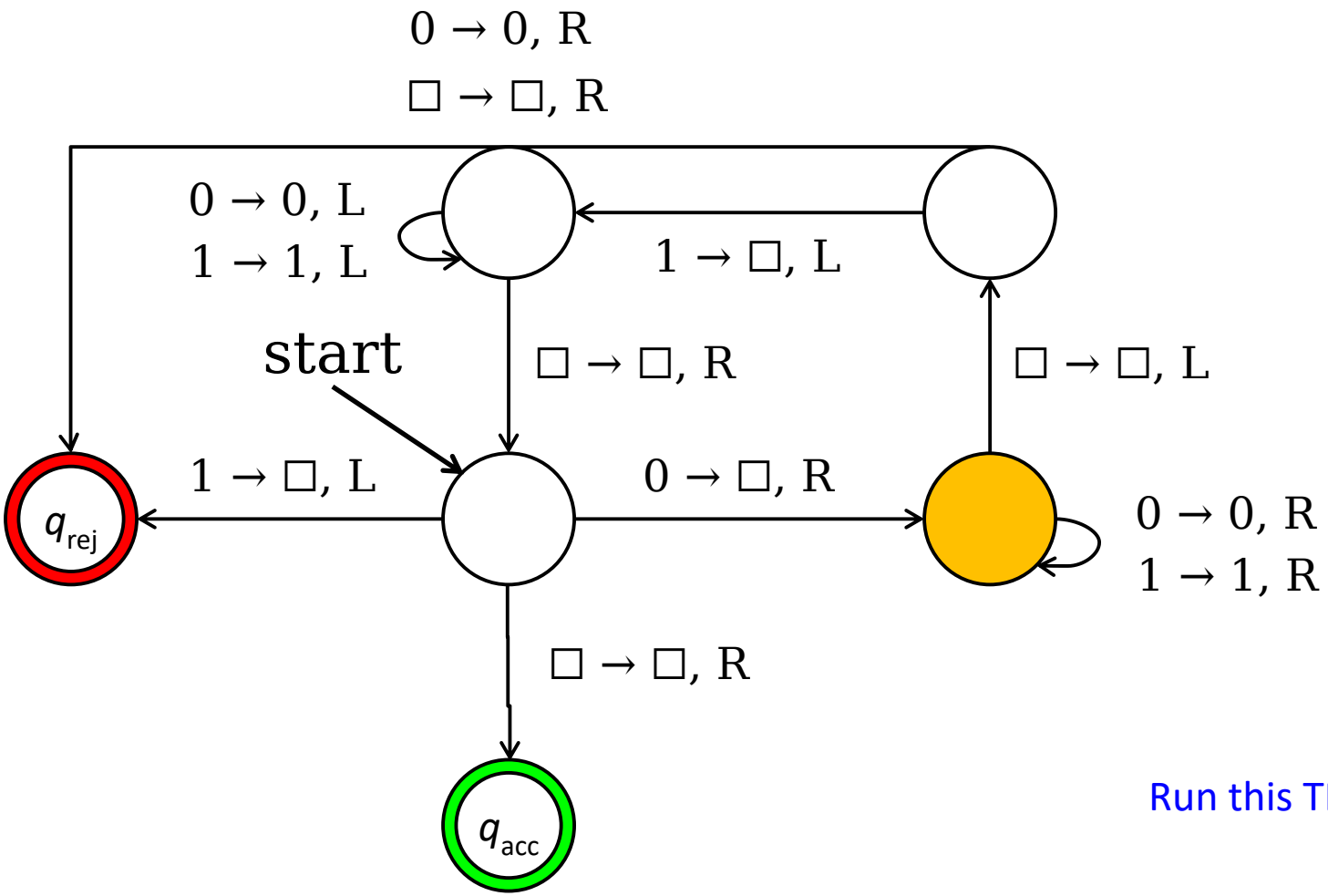
Run this TM for fifteen steps.



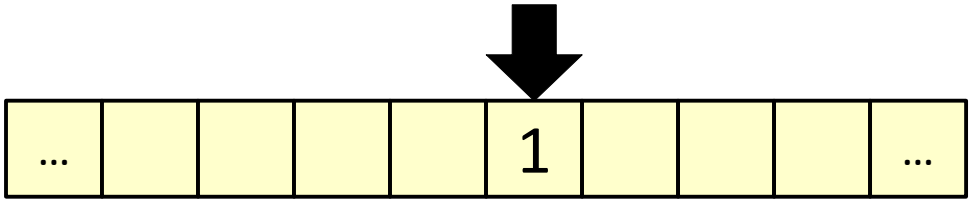


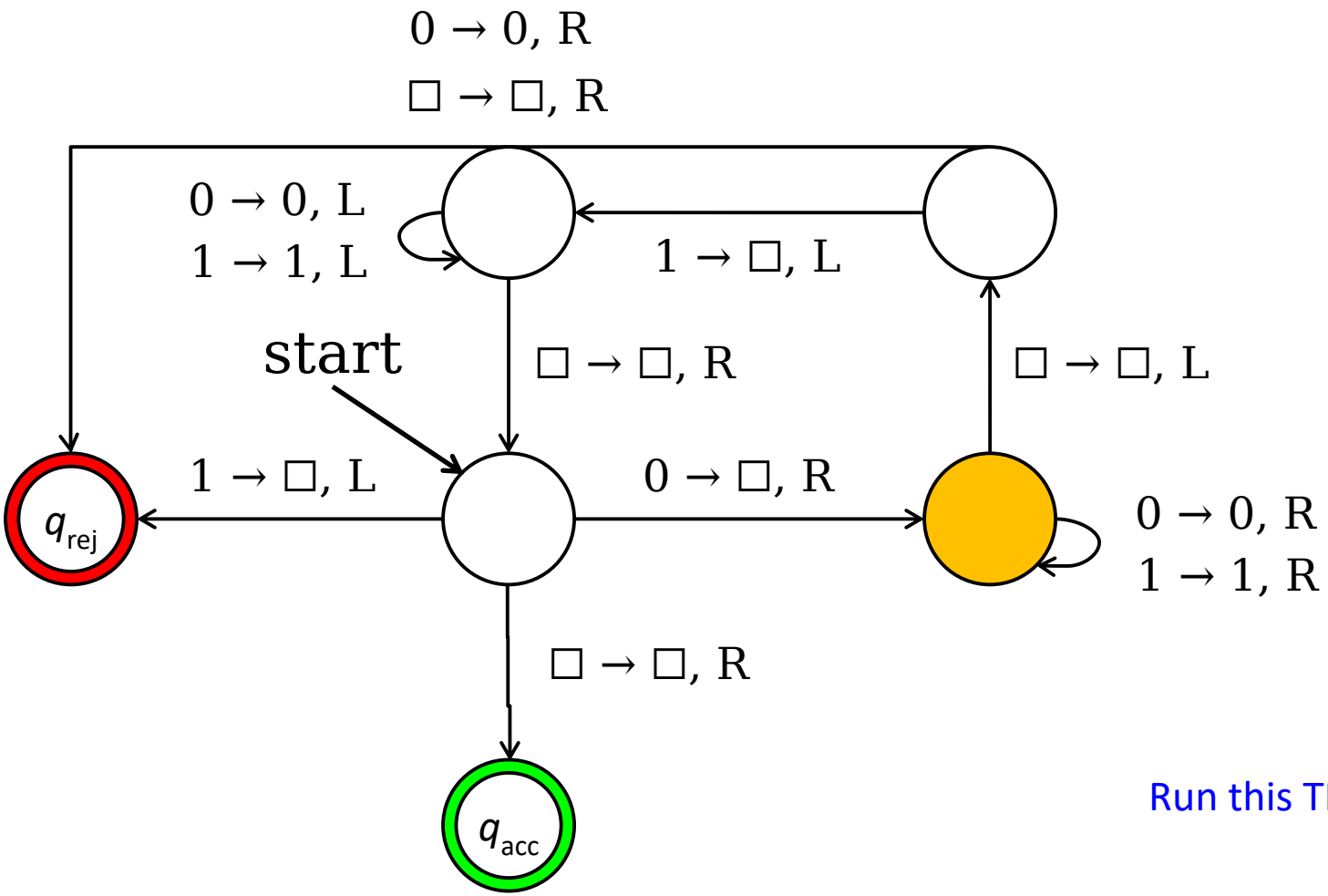
Run this TM for fifteen steps.



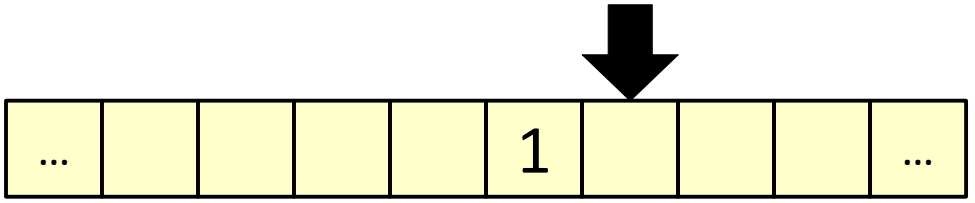


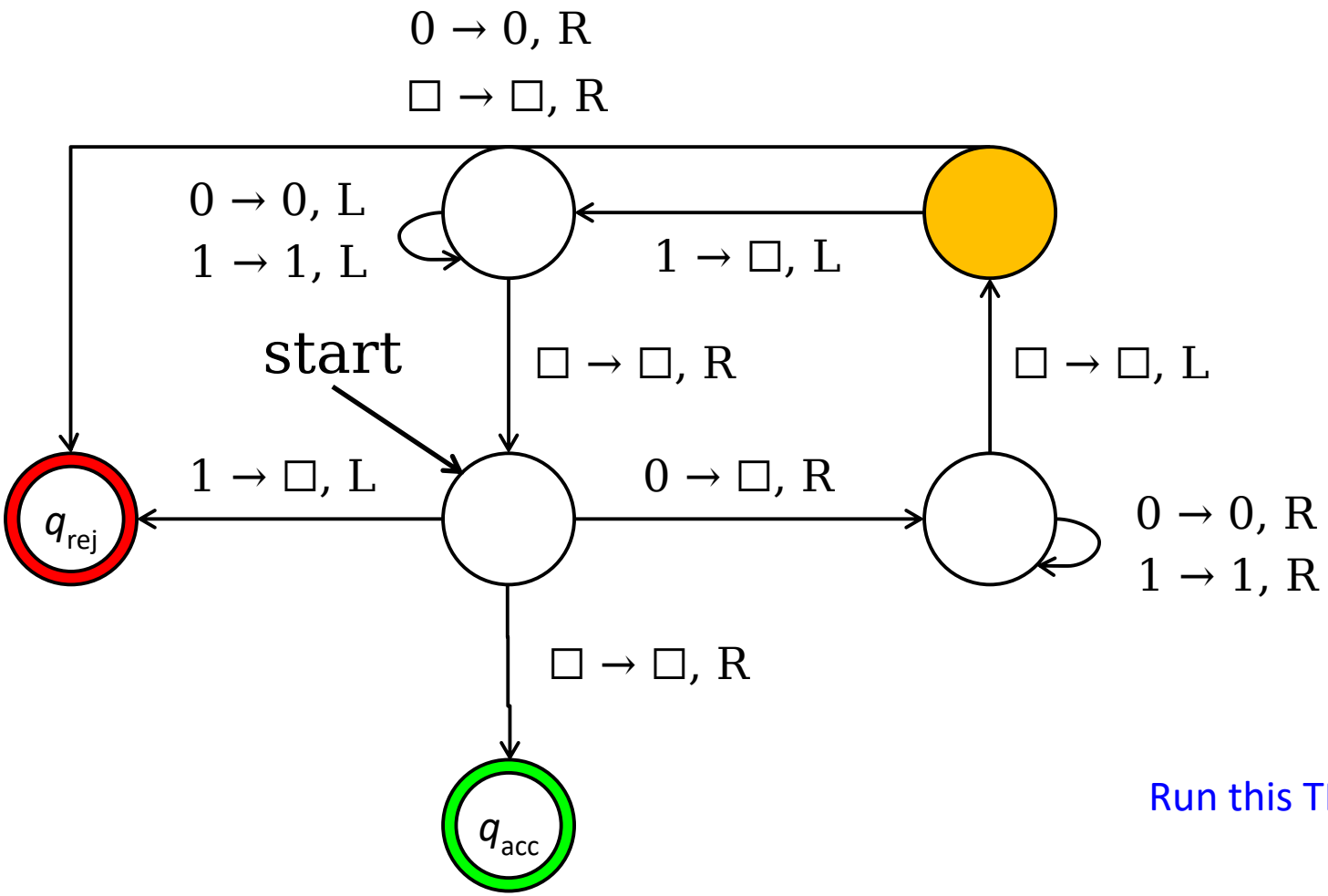
Run this TM for fifteen steps.



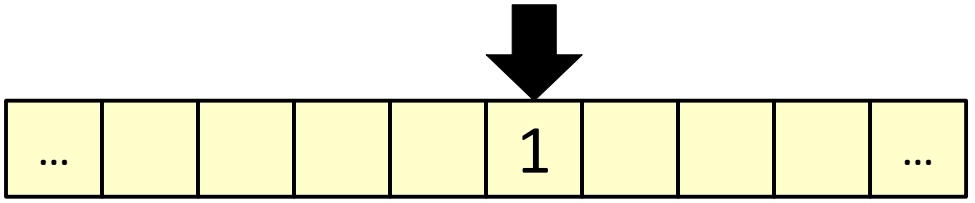


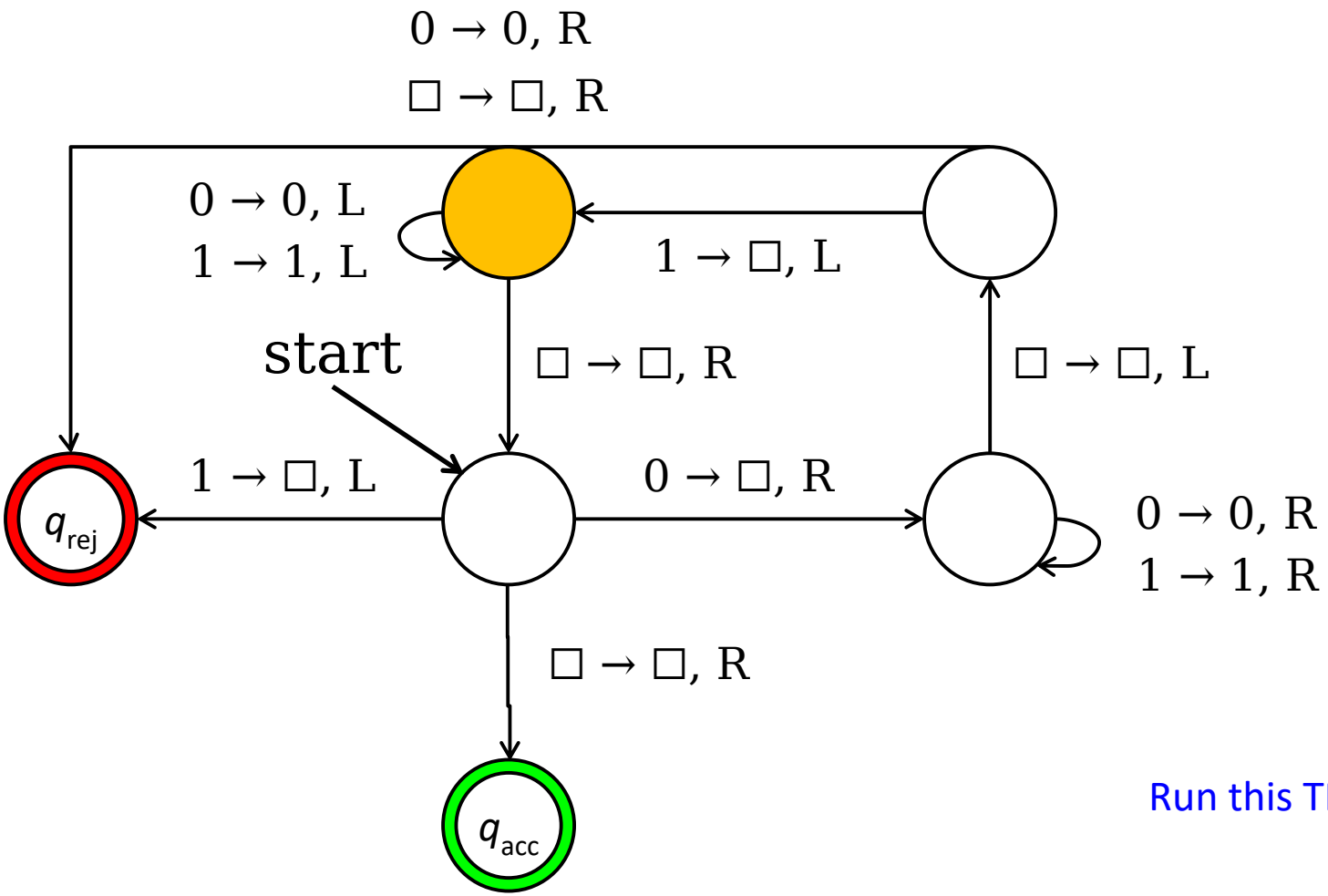
Run this TM for fifteen steps.



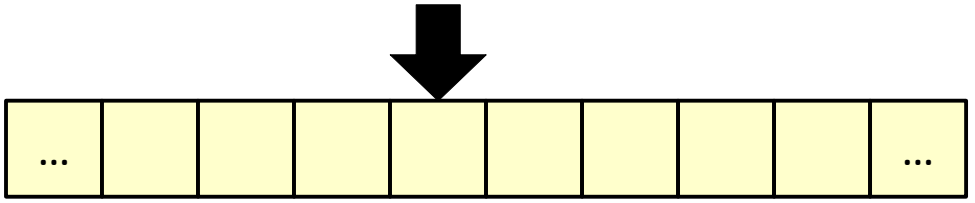


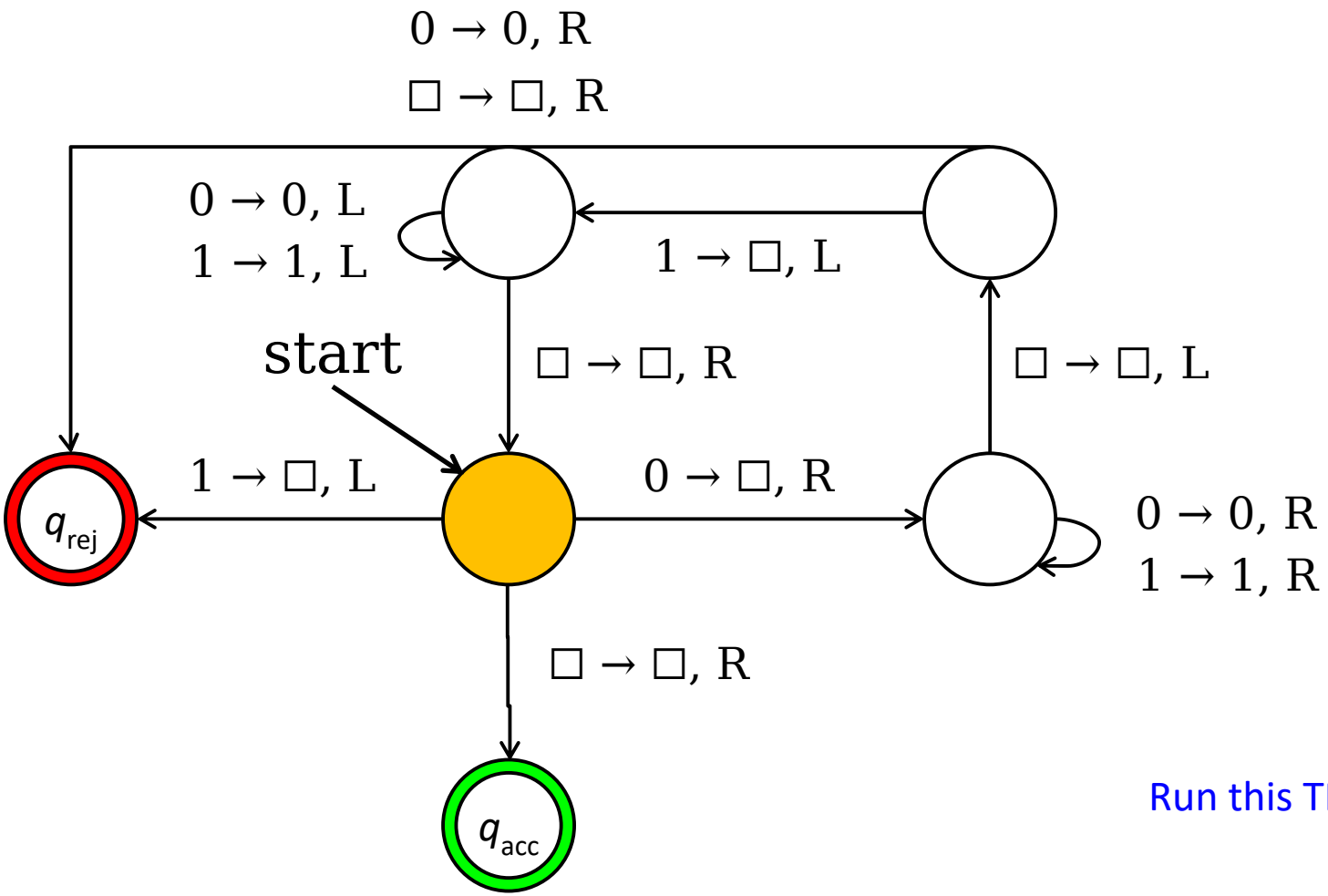
Run this TM for fifteen steps.



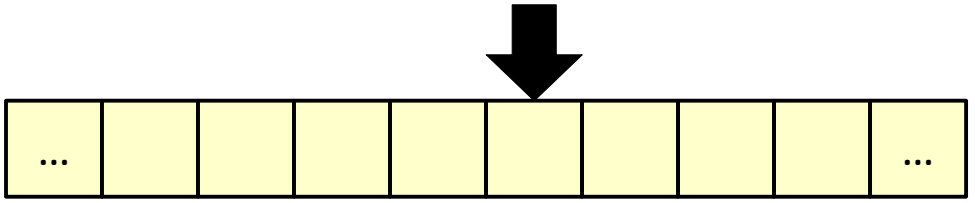


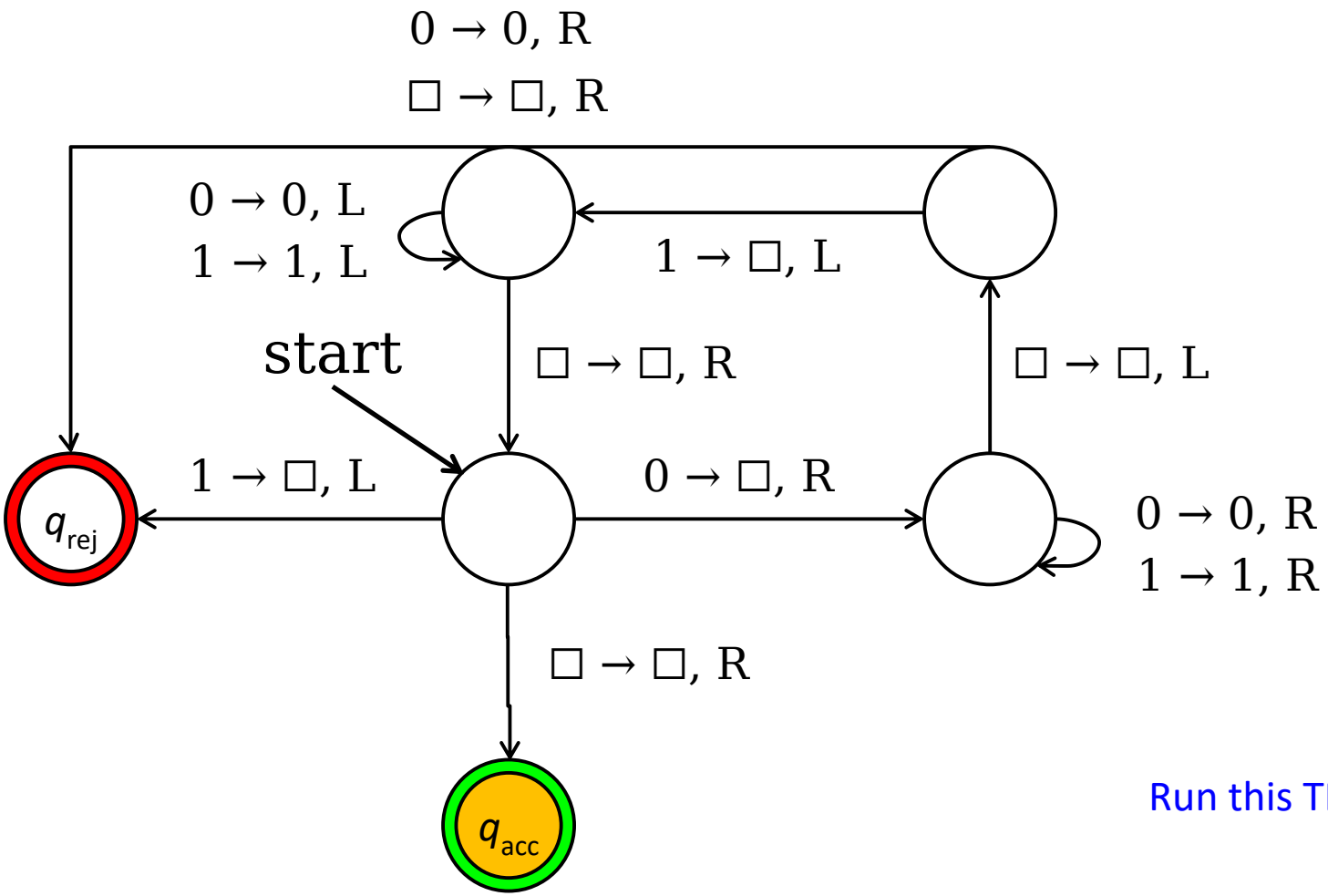
Run this TM for fifteen steps.



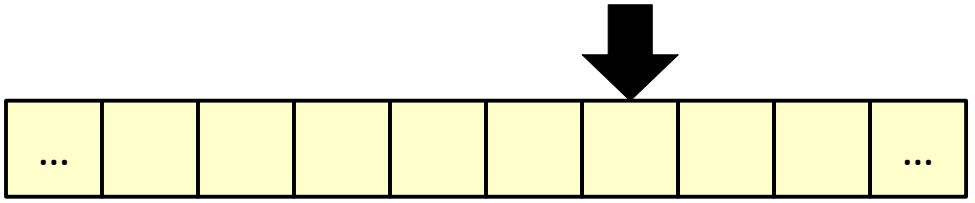


Run this TM for fifteen steps.





Run this TM for fifteen steps.



Some Verifiers

Consider A_{TM}:

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

```
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;
}
```

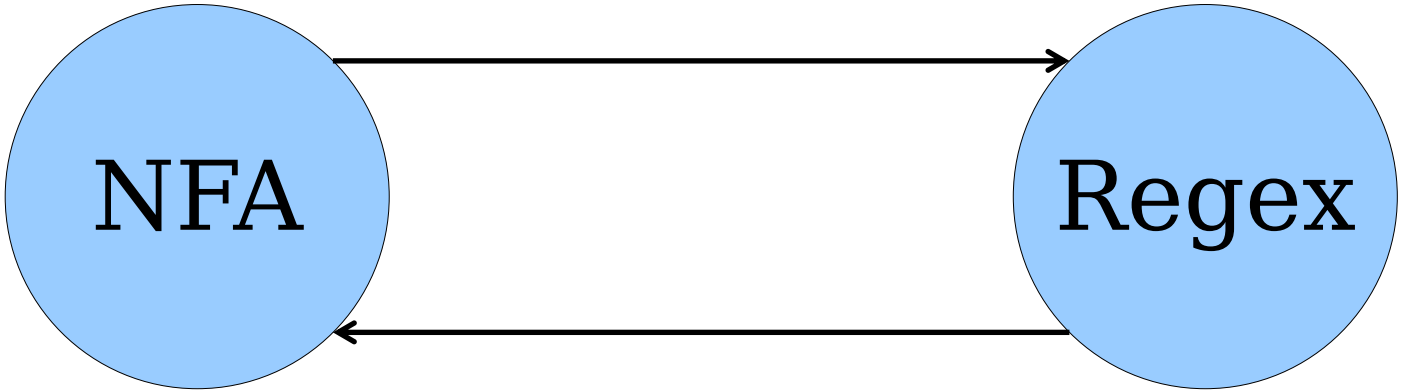
Do you see why M accepts w iff there is some c such that checkWillAccept(M, w, c) returns true?

Do you see why checkWillAccept always halts?

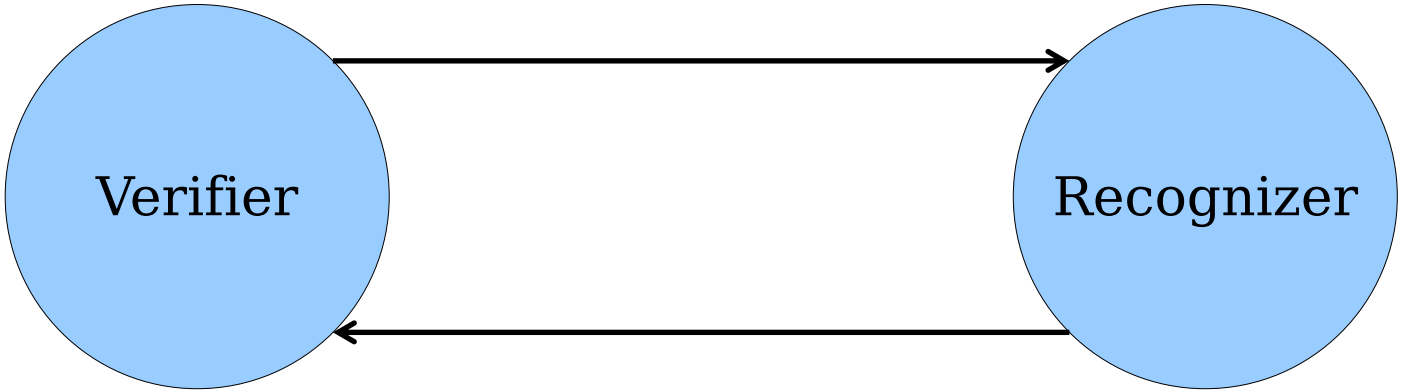
What languages are verifiable?

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.

Requirements on a verifier V for L:

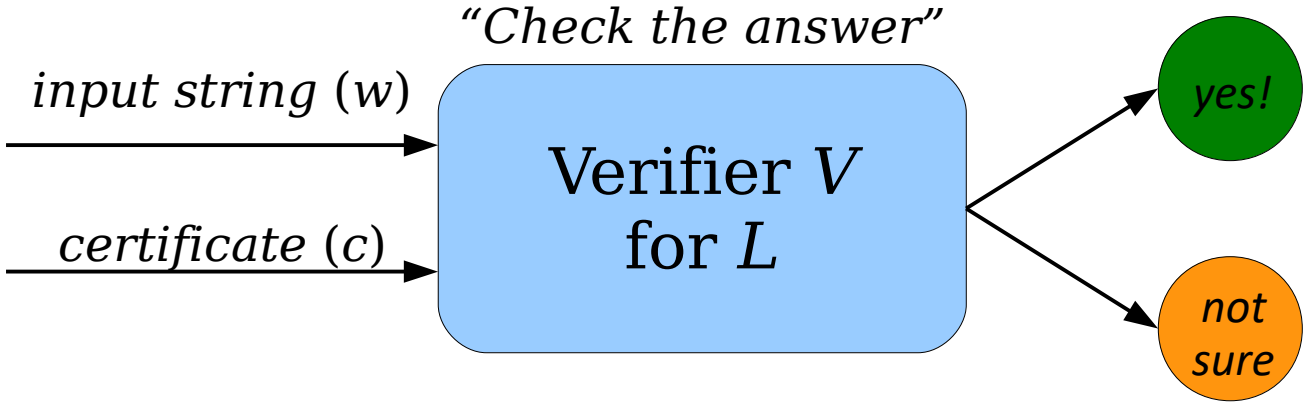
V halts on all inputs.
$\forall w \in \Sigma^*. (w \in L \leftrightarrow \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

Requirements on a recognizer M for L:

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

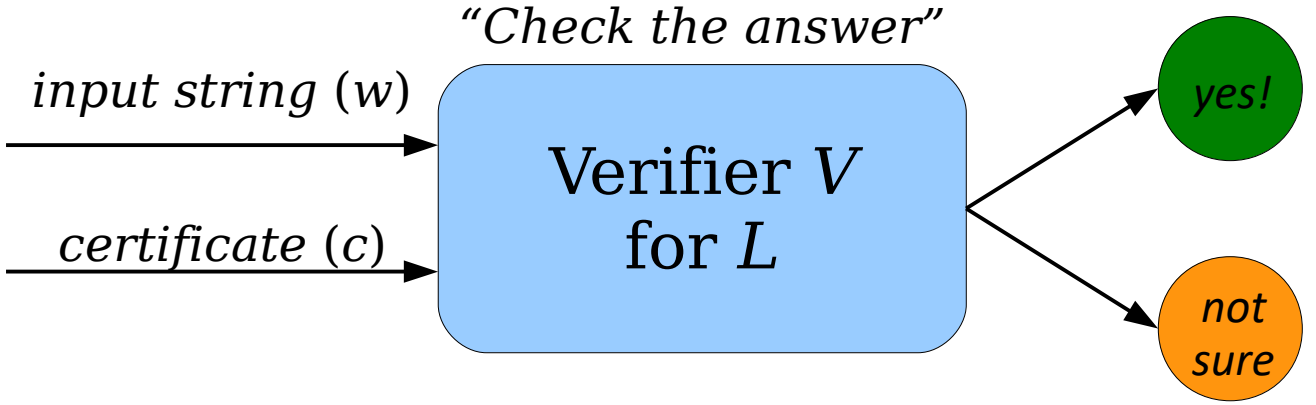
Verifiers and **RE**

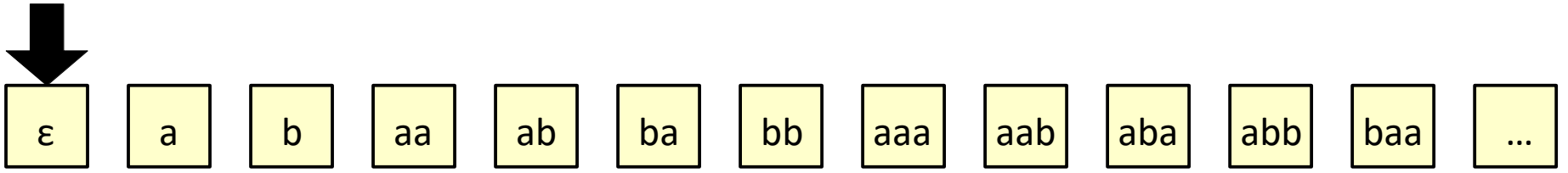
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.



Verifiers and **RE**

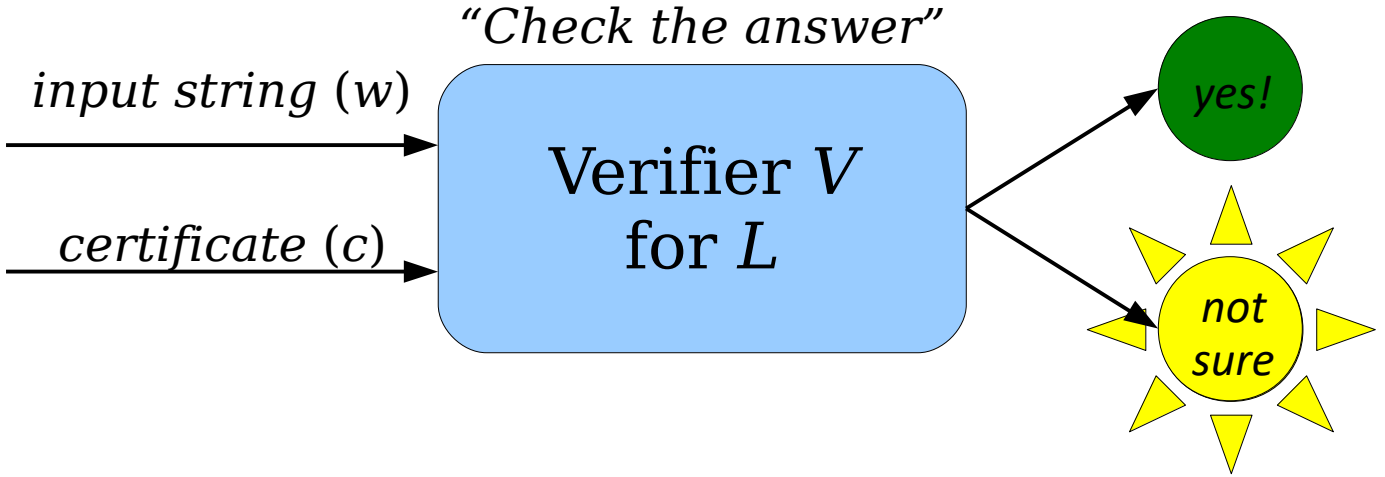
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

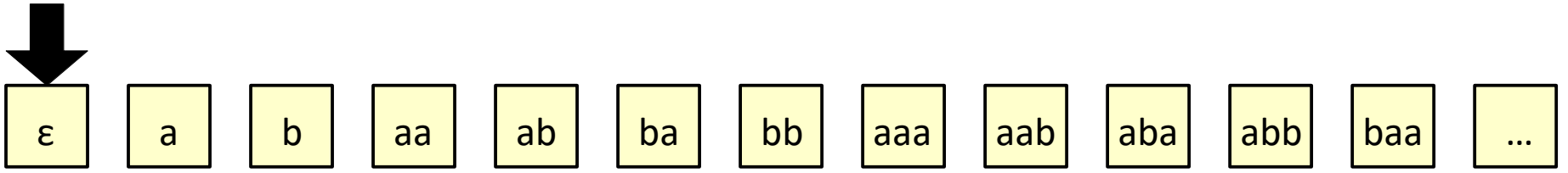




Verifiers and **RE**

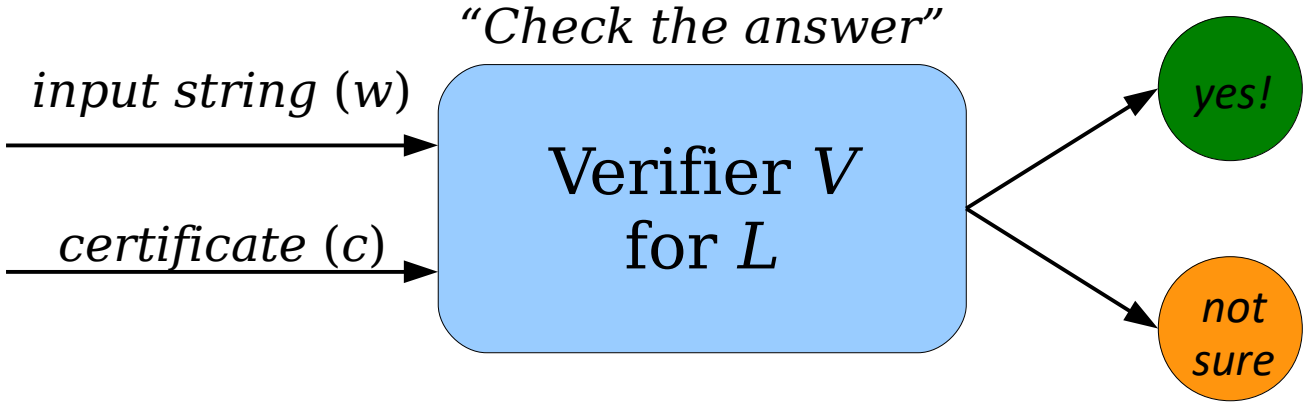
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

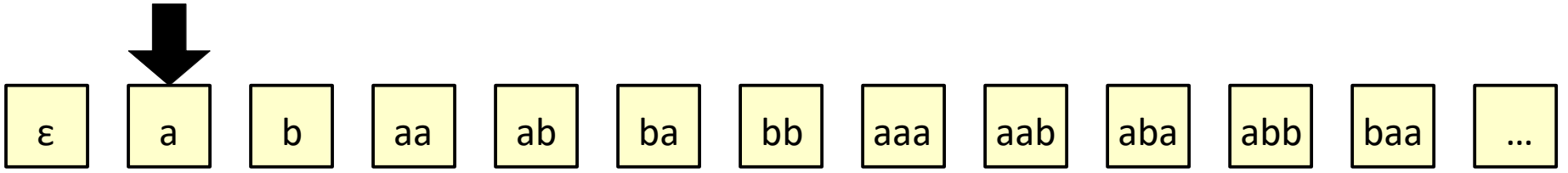




Verifiers and **RE**

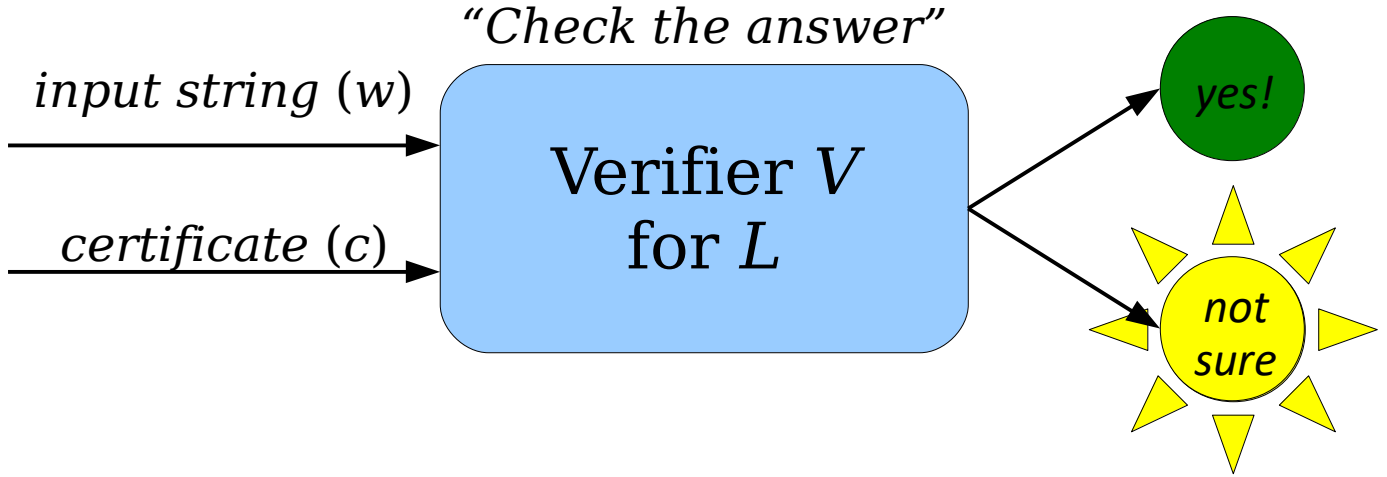
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

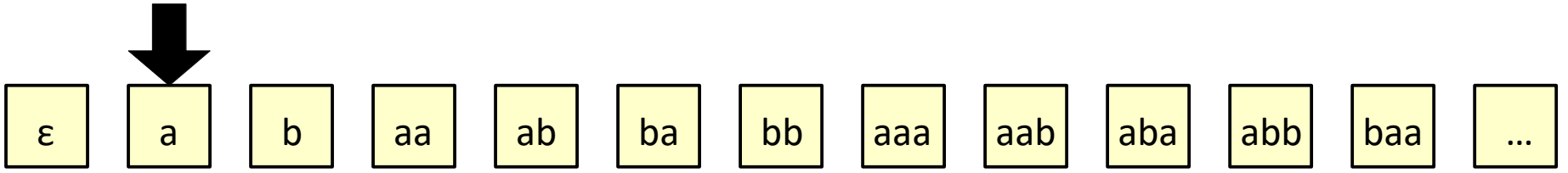




Verifiers and **RE**

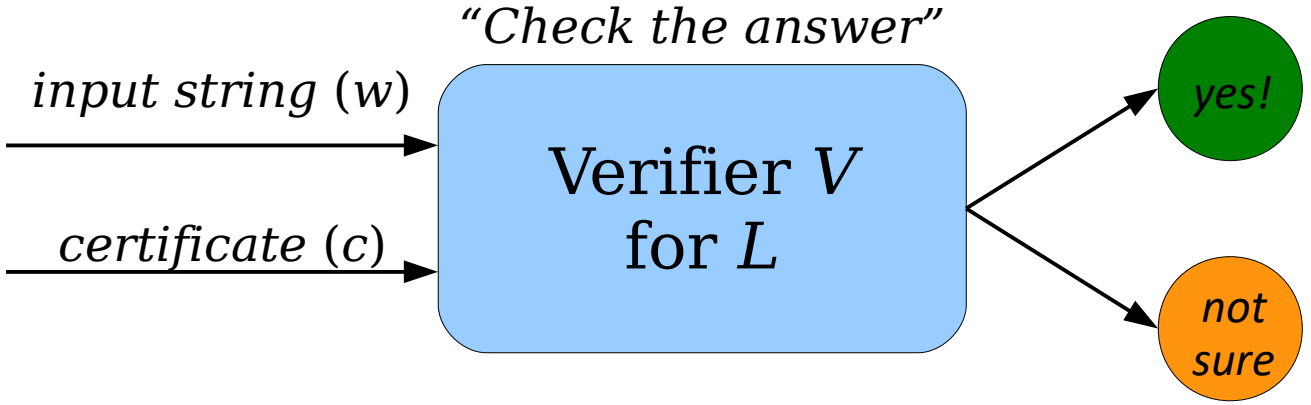
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

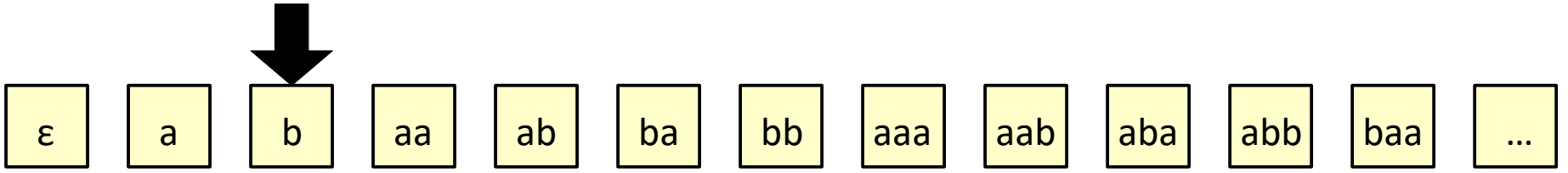




Verifiers and **RE**

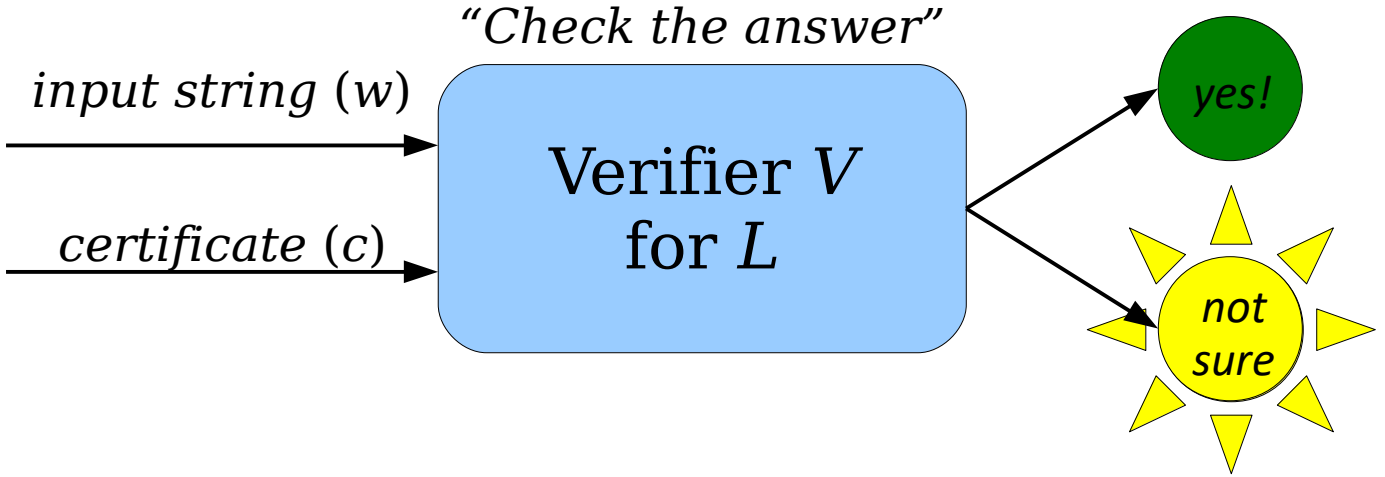
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

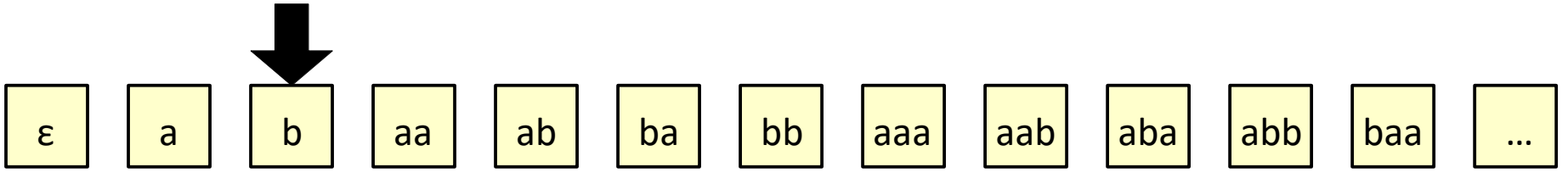




Verifiers and **RE**

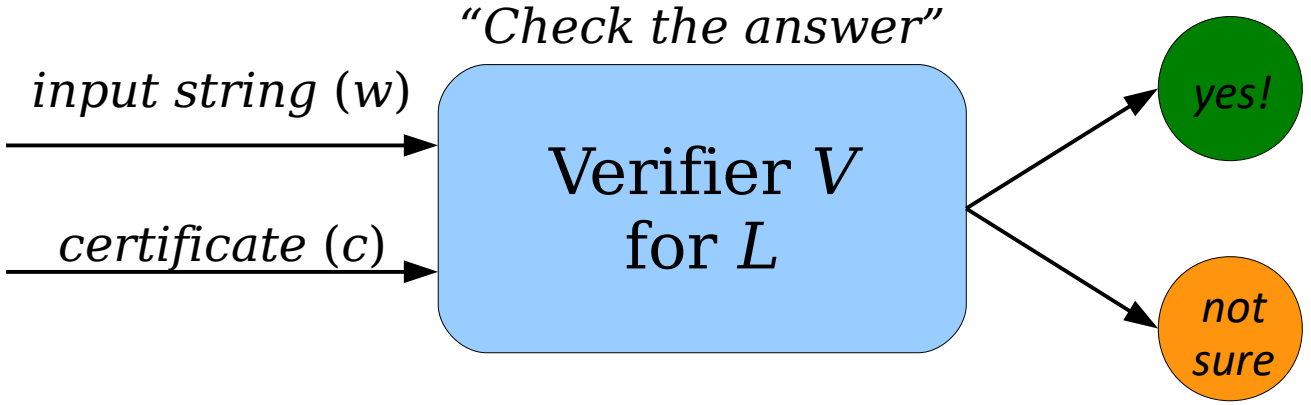
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

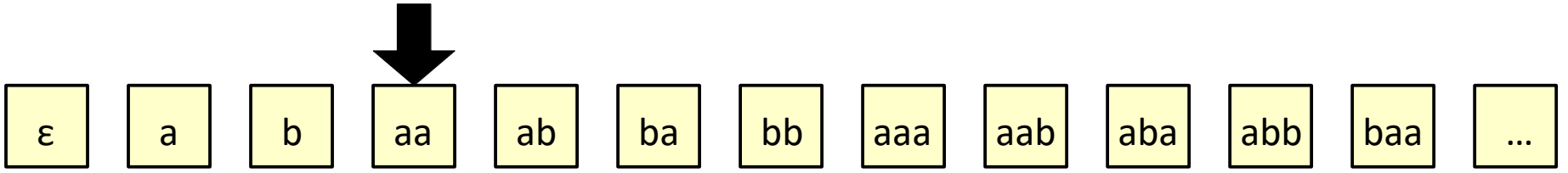




Verifiers and **RE**

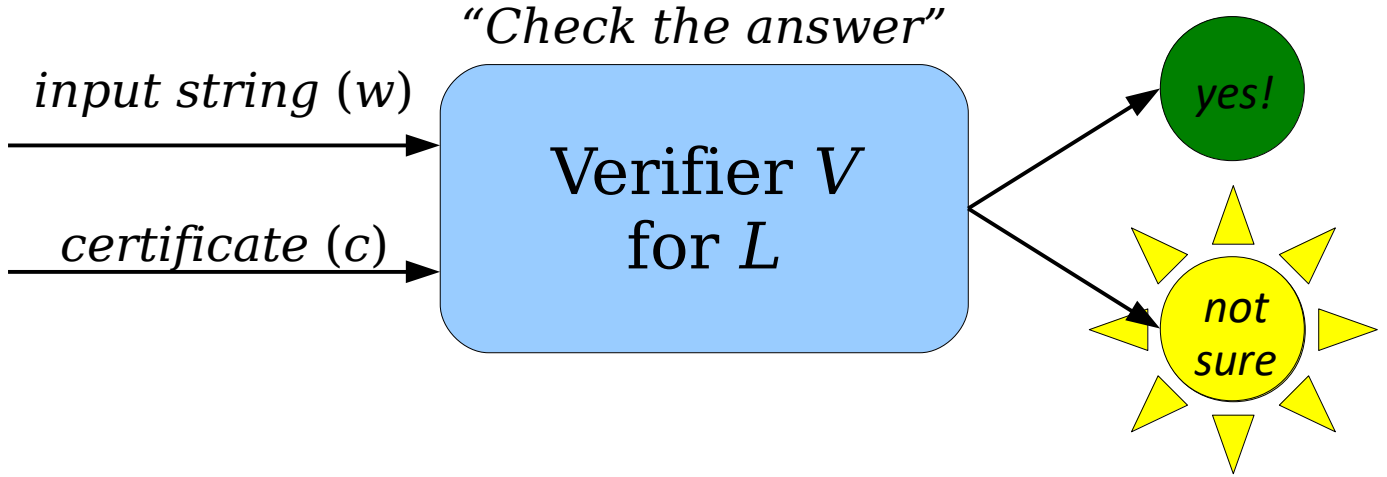
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

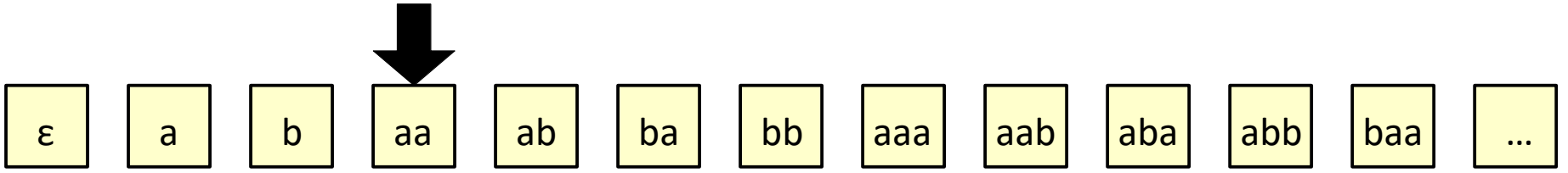




Verifiers and **RE**

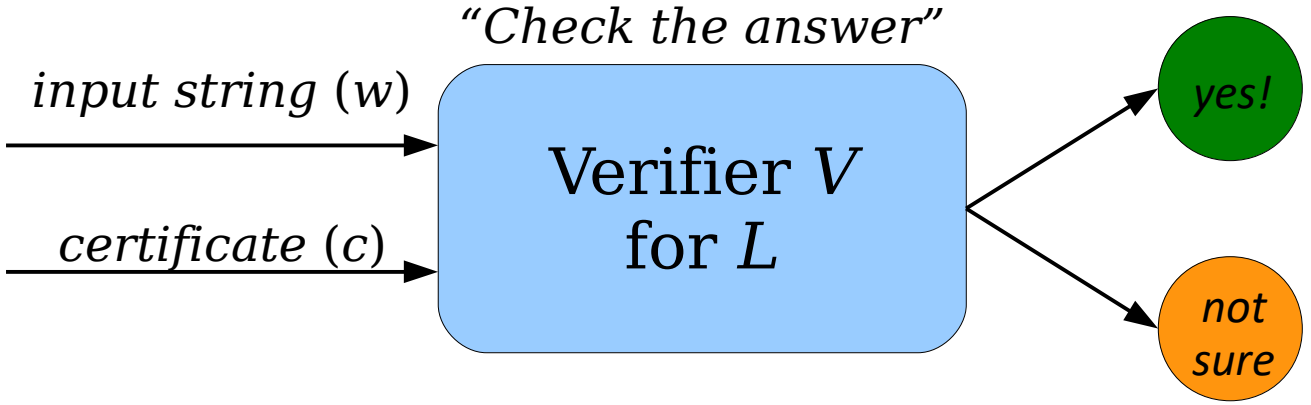
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

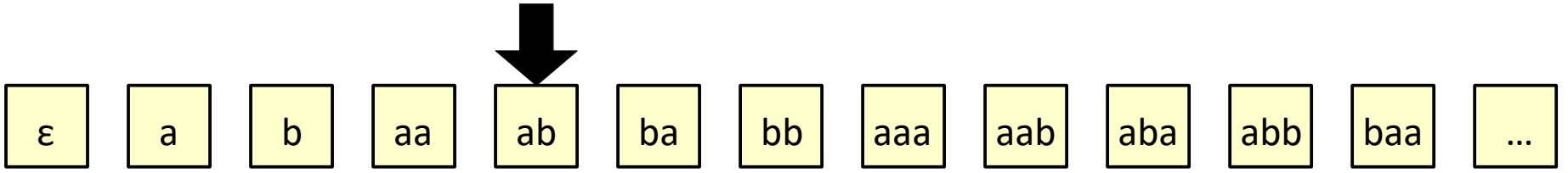




Verifiers and **RE**

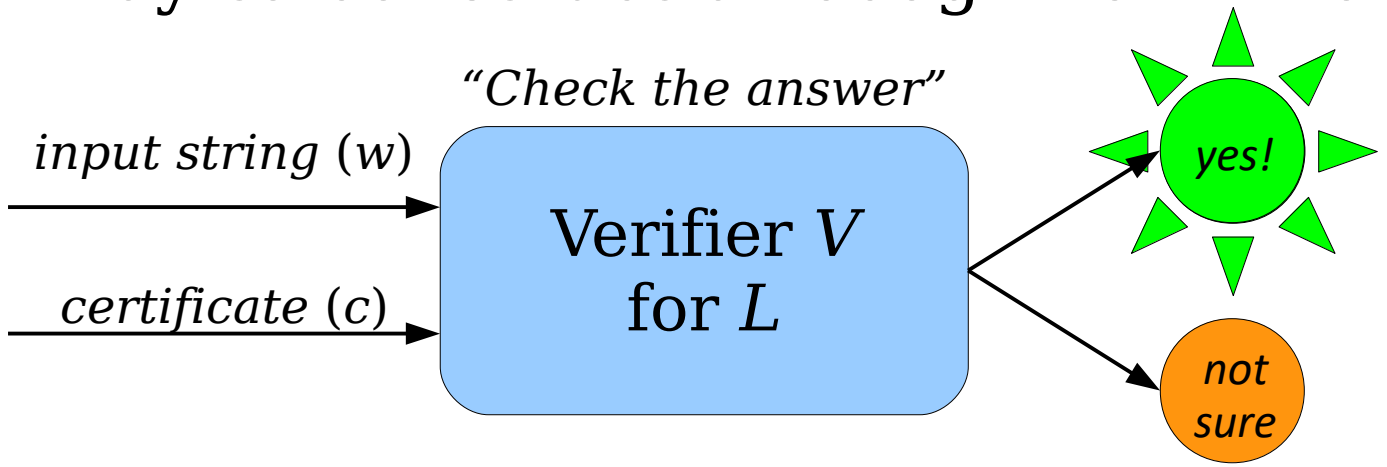
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

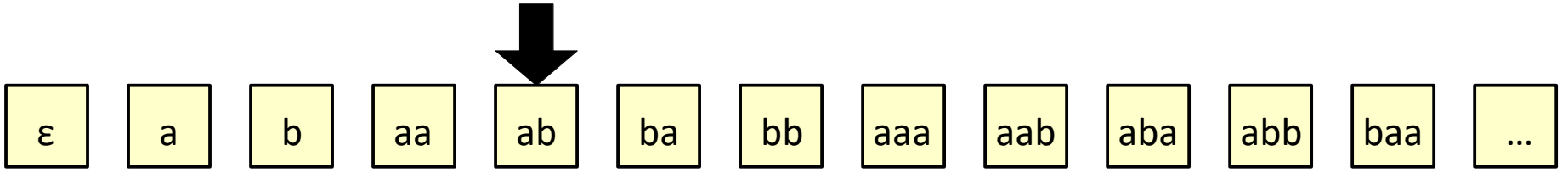




Verifiers and **RE**

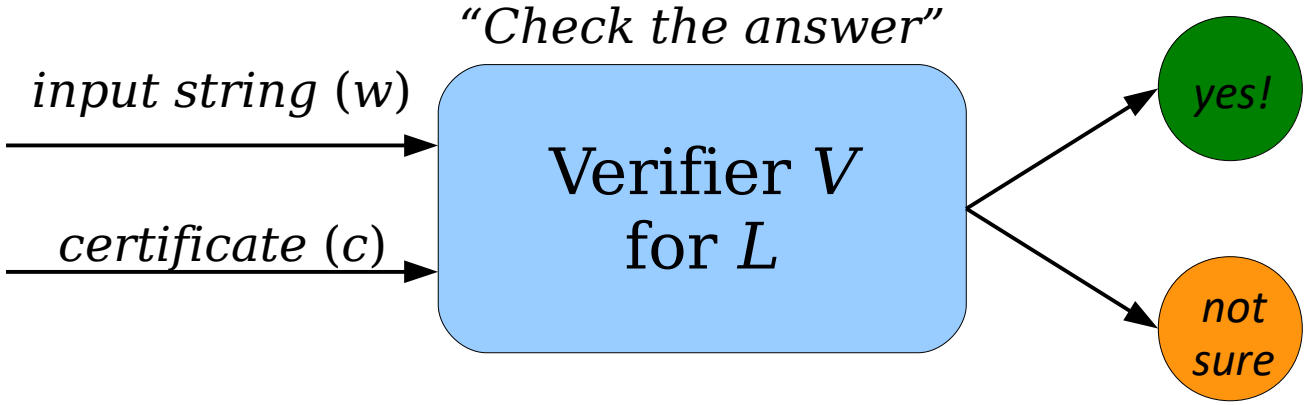
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.





Verifiers and **RE**

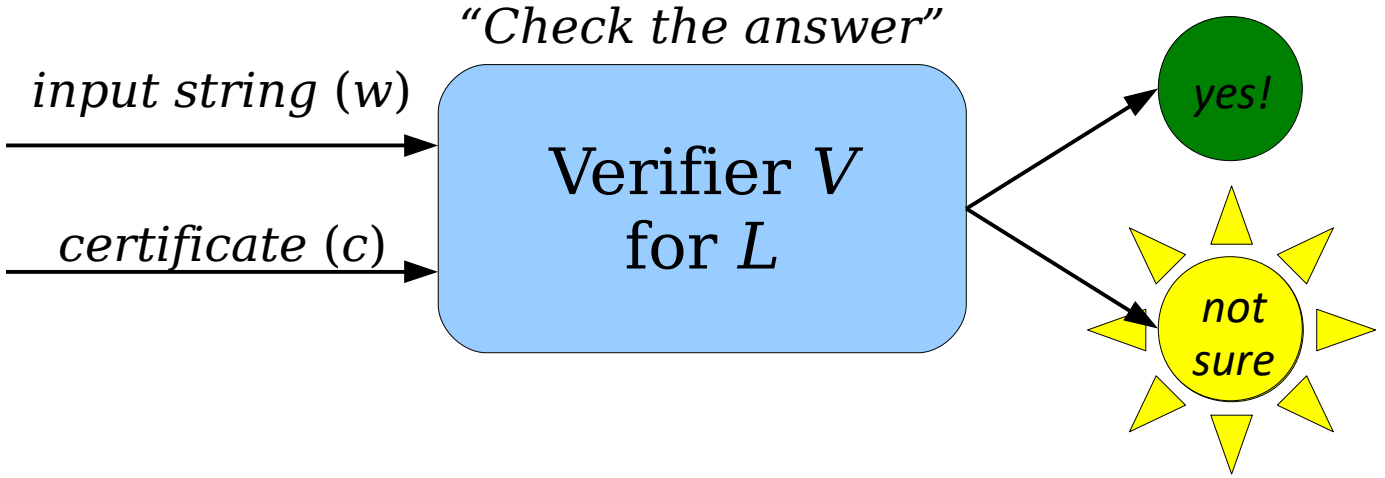
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

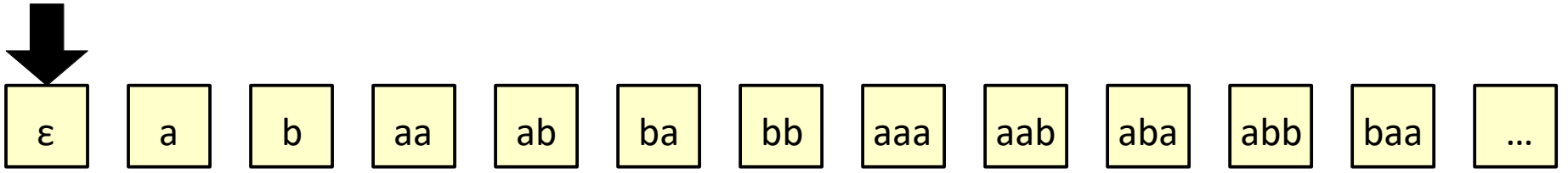


ε	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	...

Verifiers and **RE**

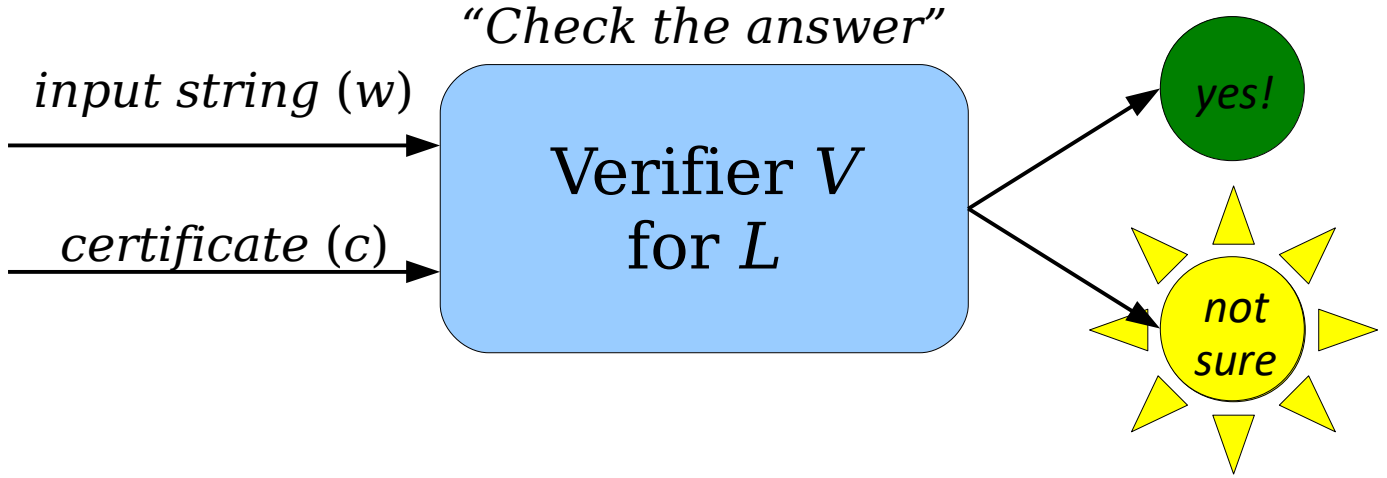
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

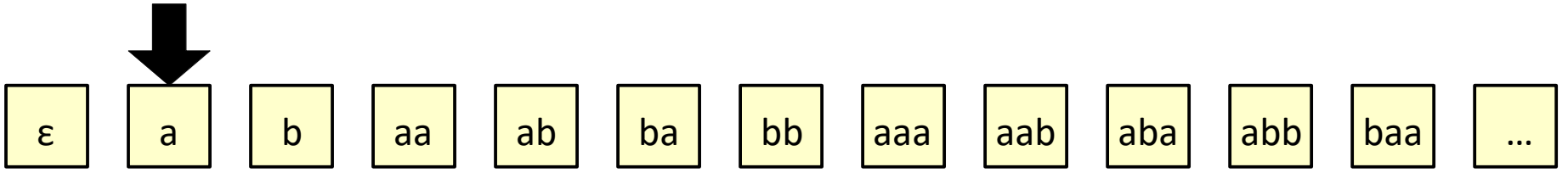




Verifiers and **RE**

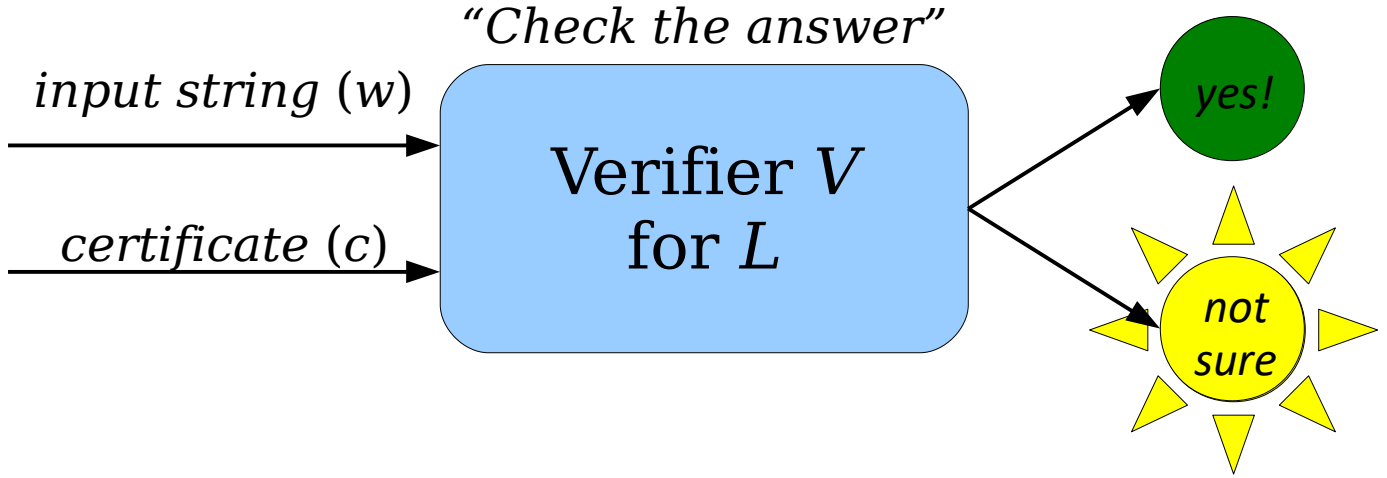
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

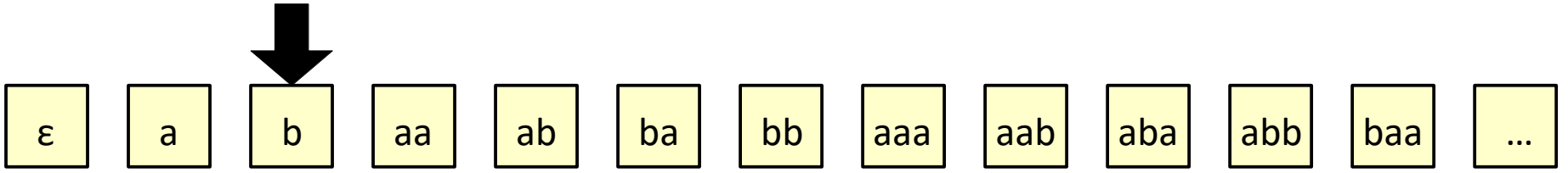




Verifiers and **RE**

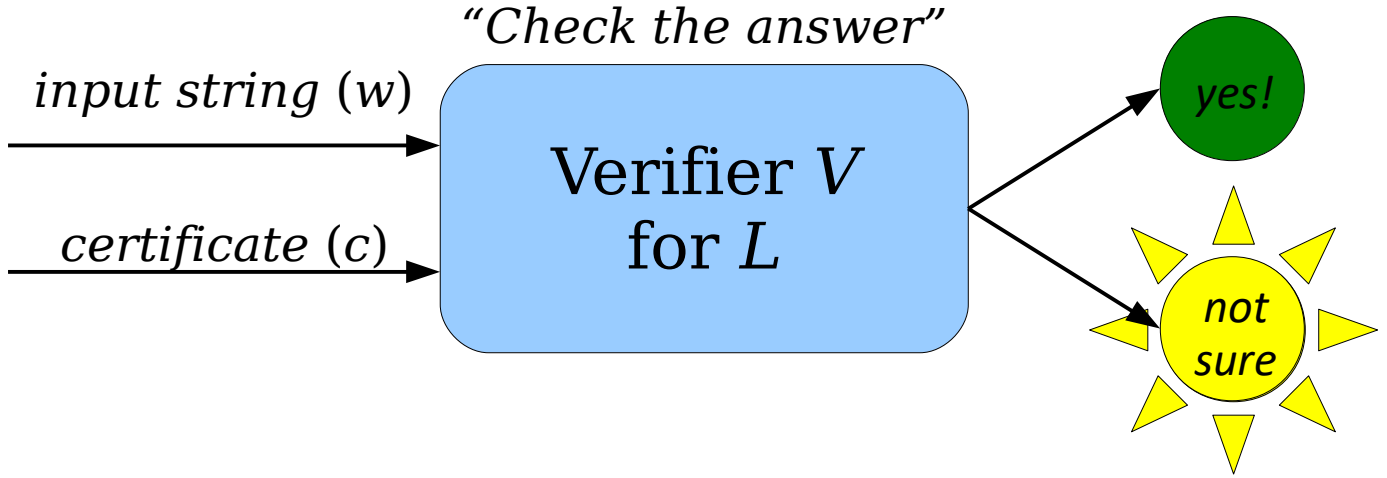
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

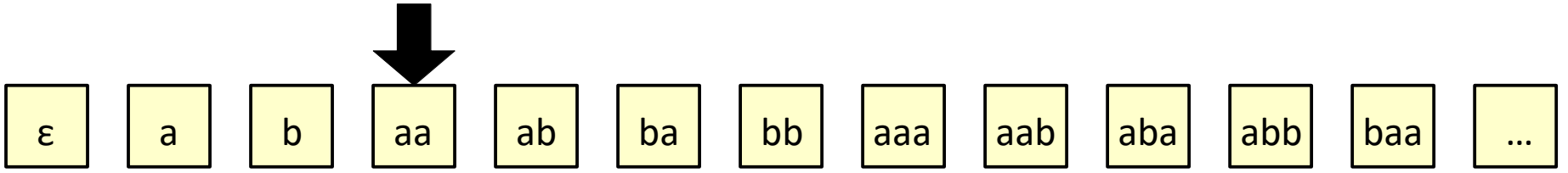




Verifiers and **RE**

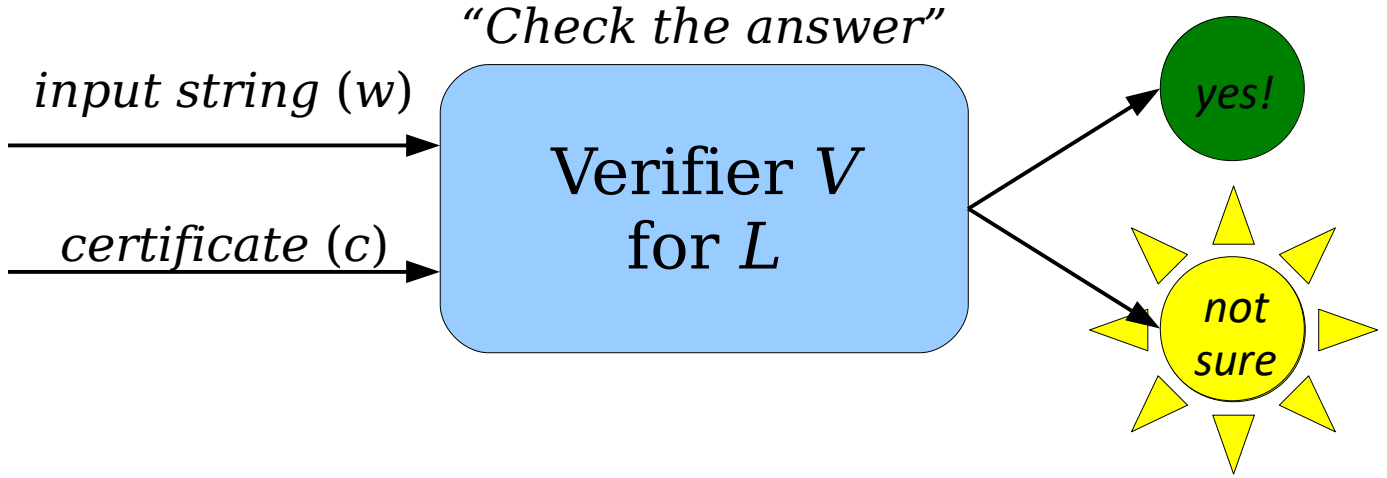
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

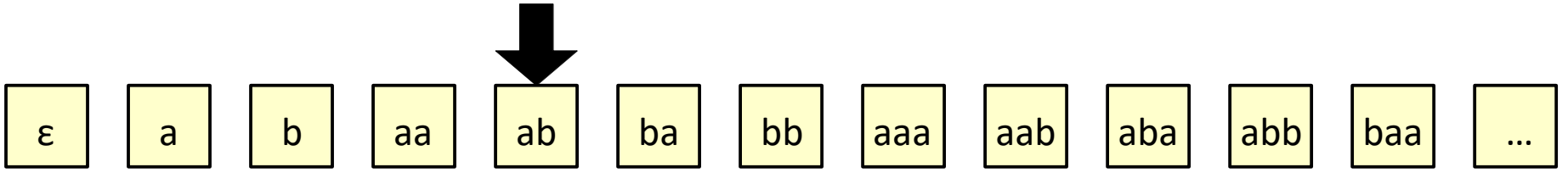




Verifiers and **RE**

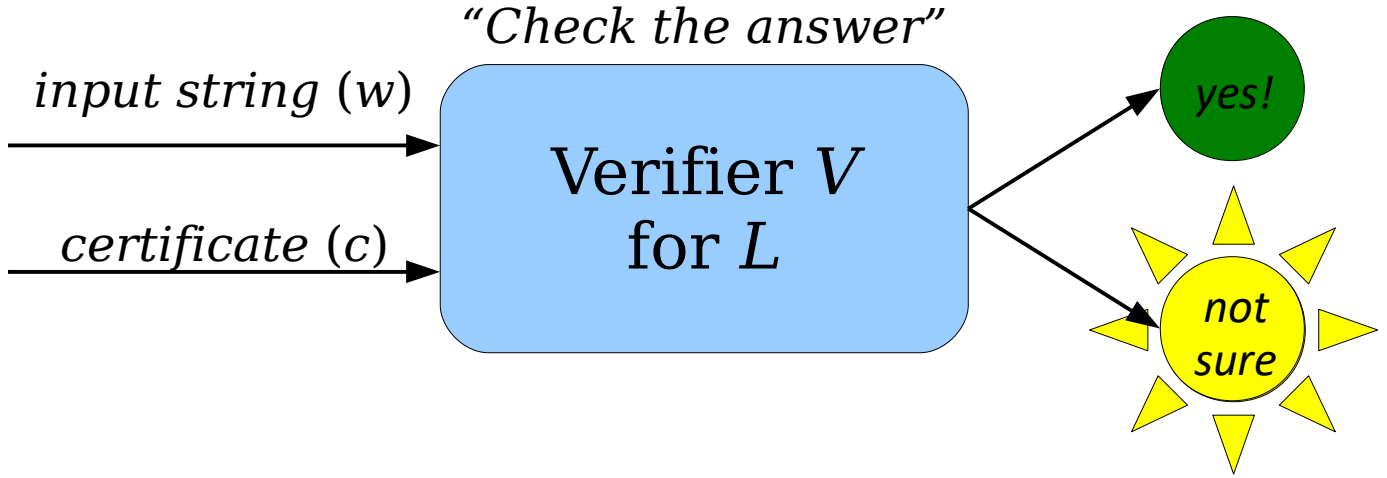
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

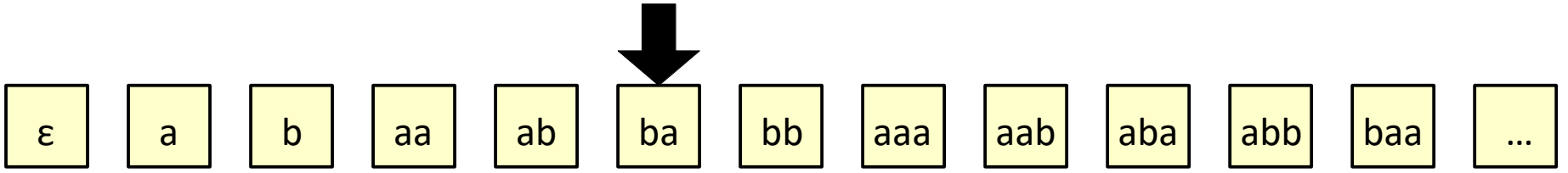




Verifiers and **RE**

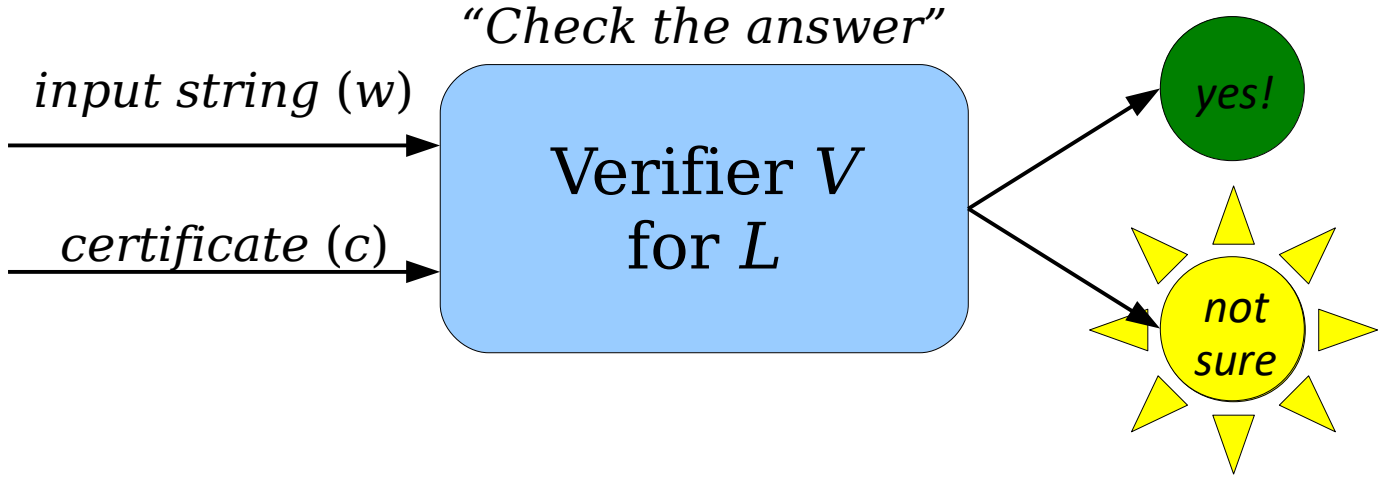
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

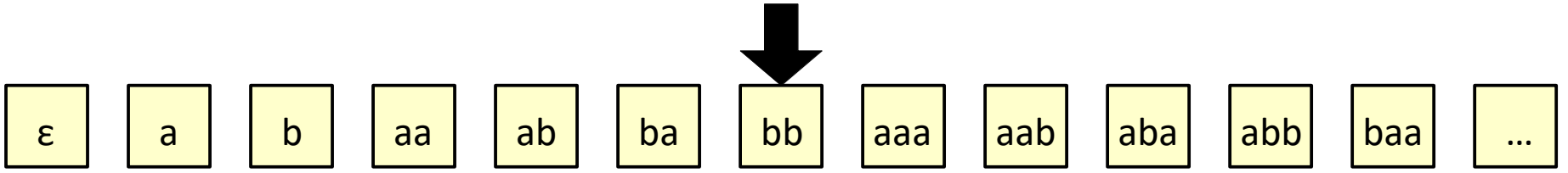




Verifiers and **RE**

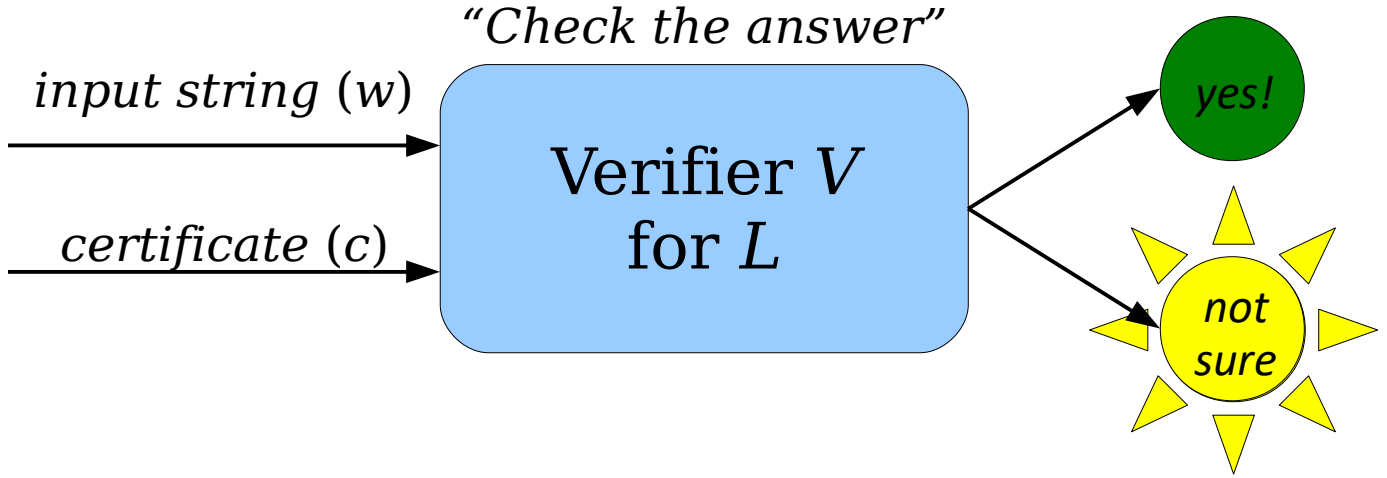
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

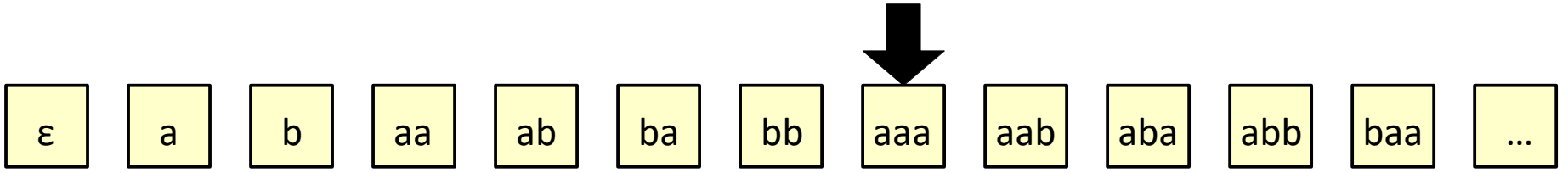




Verifiers and **RE**

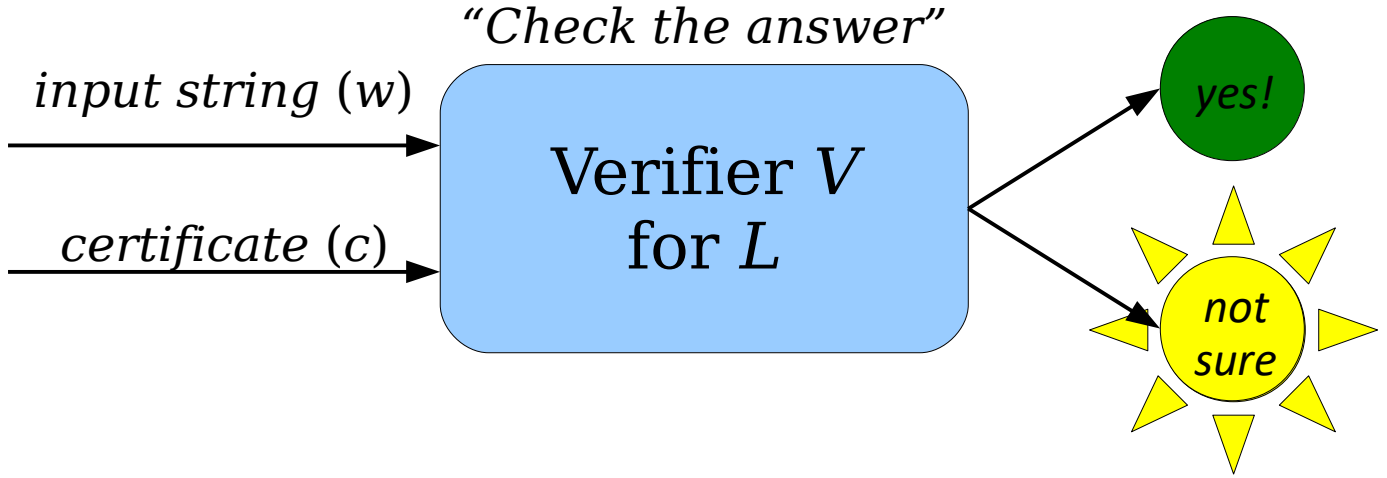
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

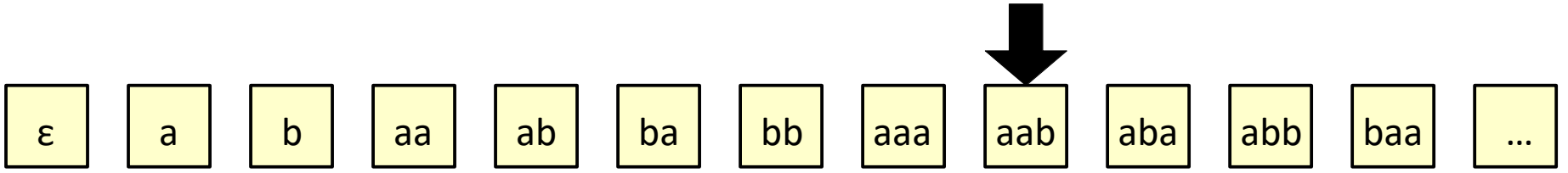




Verifiers and **RE**

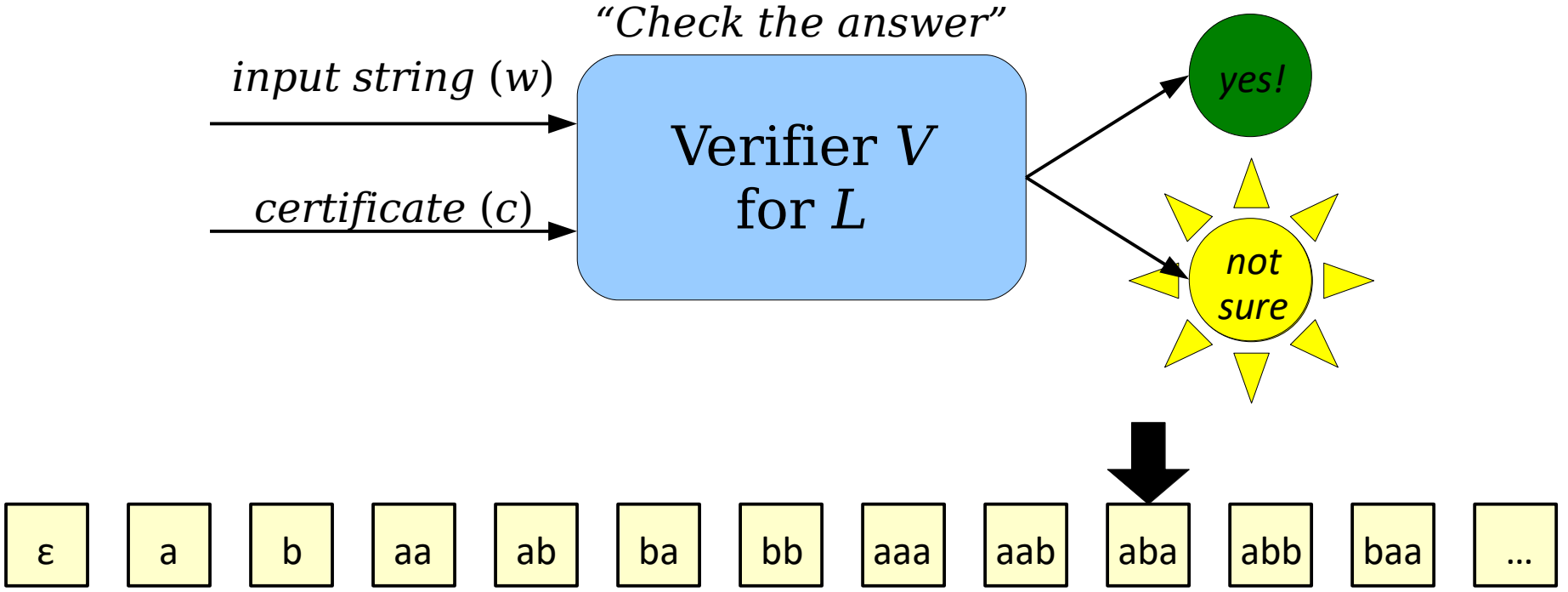
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.





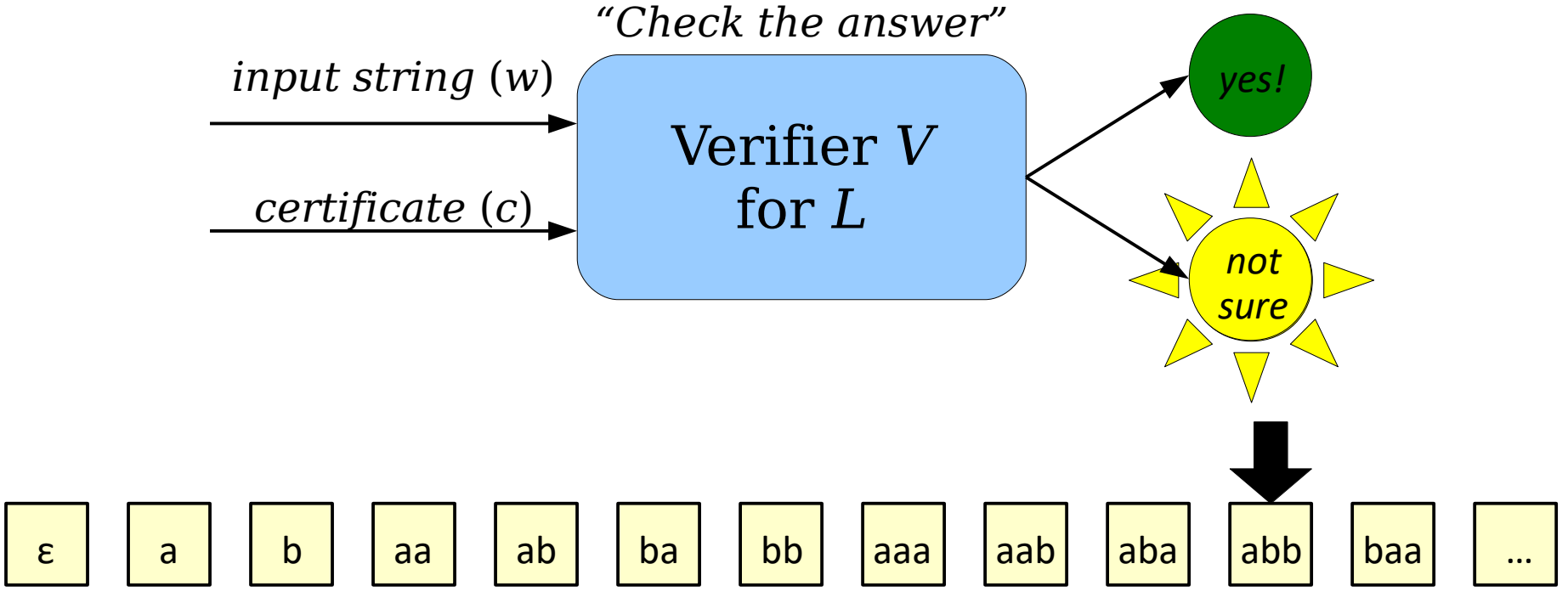
Verifiers and **RE**

- ***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.



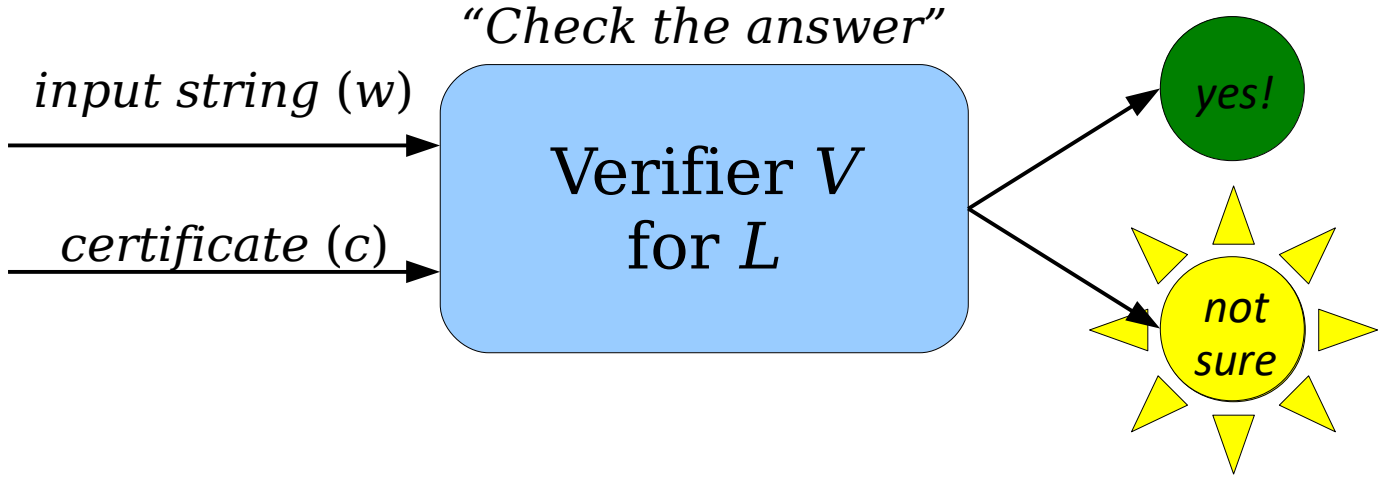
Verifiers and **RE**

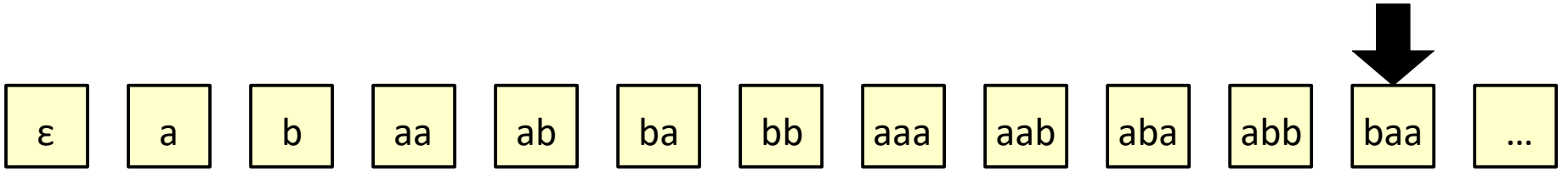
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.



Verifiers and **RE**

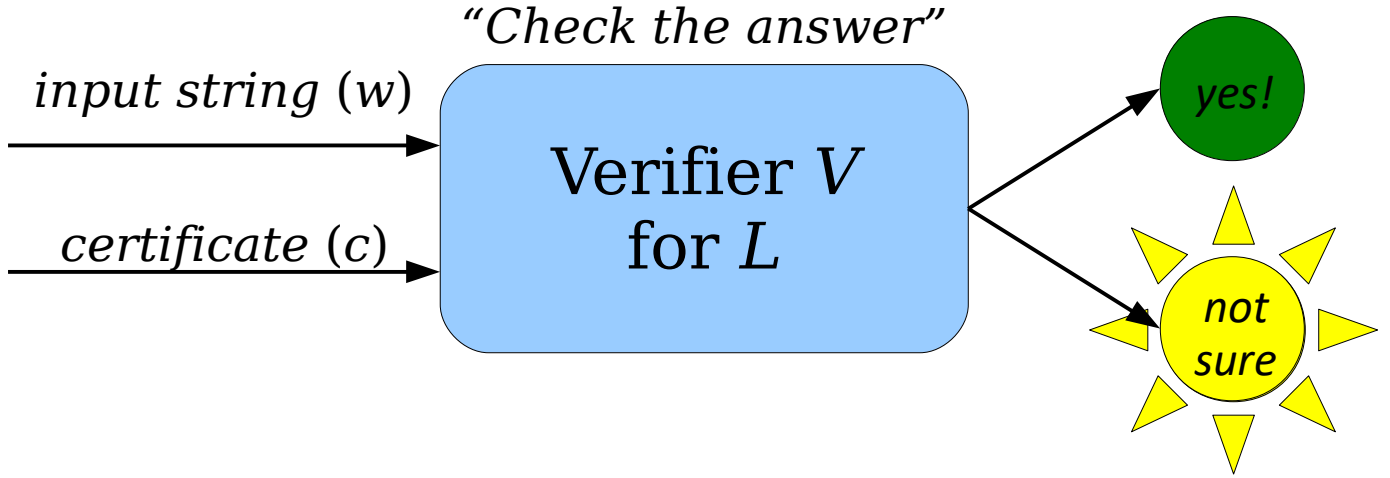
- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.

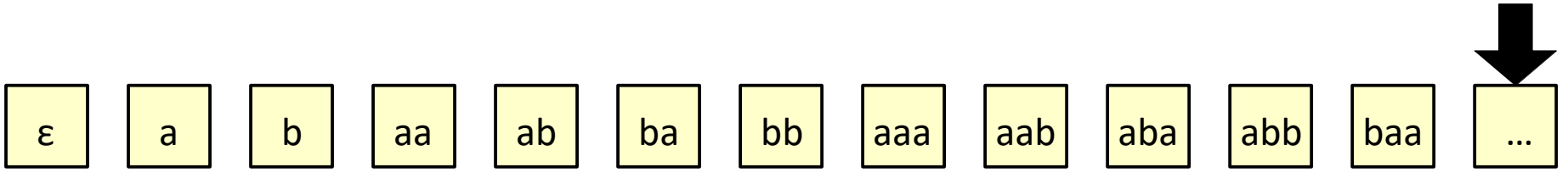




Verifiers and **RE**

- ***Theorem:*** If 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) {
  for (each string c) {
    if (V accepts ⟨w, c⟩) return true;
  }
}
```

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 certificates, 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.

Requirements on a recognizer M for L:

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

Requirements on a verifier V for L:

V halts on all inputs.
$\forall w \in \Sigma^*. (w \in L \leftrightarrow \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

We have a recognizer for a language.

We want to turn it into a verifier.

Where did we see this before?

Some Verifie

Observation: This trick of enforcing a step count limits how long M can run for!

Consider A_{TM}:

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

```
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 checkWillAccept always halts?

Verifiers and **RE**

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

Proof sketch: Let L be a **RE** language and let M be a recognizer for it. Consider this function:

```
bool checkIsInL(string w, int c) {
   TM M = /* hardcoded version of a recognizer for L */;
   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;
}
```

Note 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 Sets

- Problem Set Five was due yesterday.
- You *can* use a late period to extend the deadline to Saturday night.
- Problem Set Six goes out later today. It's due next *Wednesday* at 11:59PM.
- PS6 is shorter and designed to be able to be completed in 5 days.
- Due to university policies, ***no late submissions will be accepted for PS6***. Please budget at least two hours before the deadline to submit the assignment.

The Last Guide

We've posted the final guide on the course website:

The ***Guide to the Lava Diagram***, which provides an intuition for how different classes of languages relate to one another.

Give this a read – there's a ton of useful information in there!

Final Exam Logistics

- Our final exam is Friday, August 14th.
- The exam is cumulative. You're responsible for topics from PS0 – PS6 and all of the lectures up through and including today's.
- The exam is open-book, open-computer, and open-notes.
- Students with OAE accommodations: if we don't yet have your OAE letter, please send it to us ASAP. If you sent it for the midterm, you don't need to send it again.

Preparing for the Exam

- We've posted some additional practice problems and exams on the course website under "Extra Problems".
- We'll have some practice final exams up today.
- ***Review Session*** on Monday, August 12th here during class, led by your lovely TAs!
- ***Practice Final*** on Wednesday, August 14th from 5:30-8:30 PM upstairs in Gates 104.

No Questions Today

Have one? Go to sli.do, and input code G517, select the old event, and vote/submit!

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?

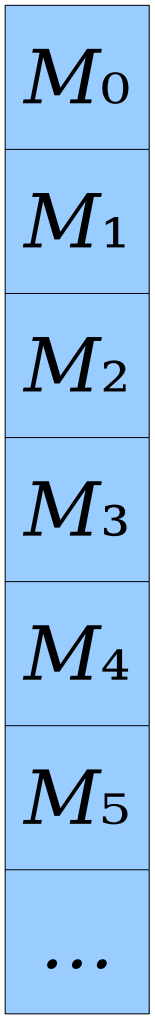
Languages, TMs, and TM Encodings

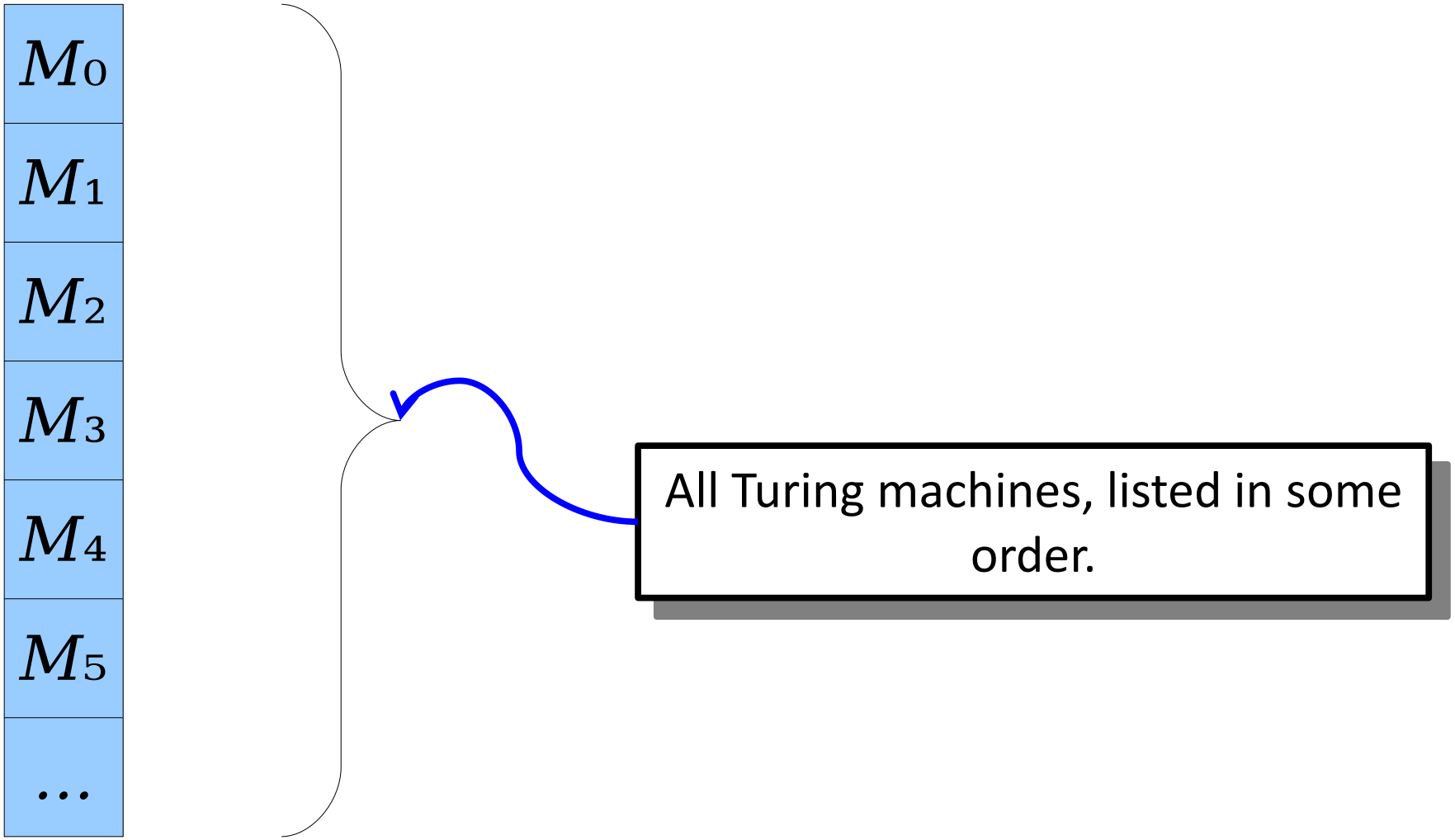
Recall: The language of a TM M is the set

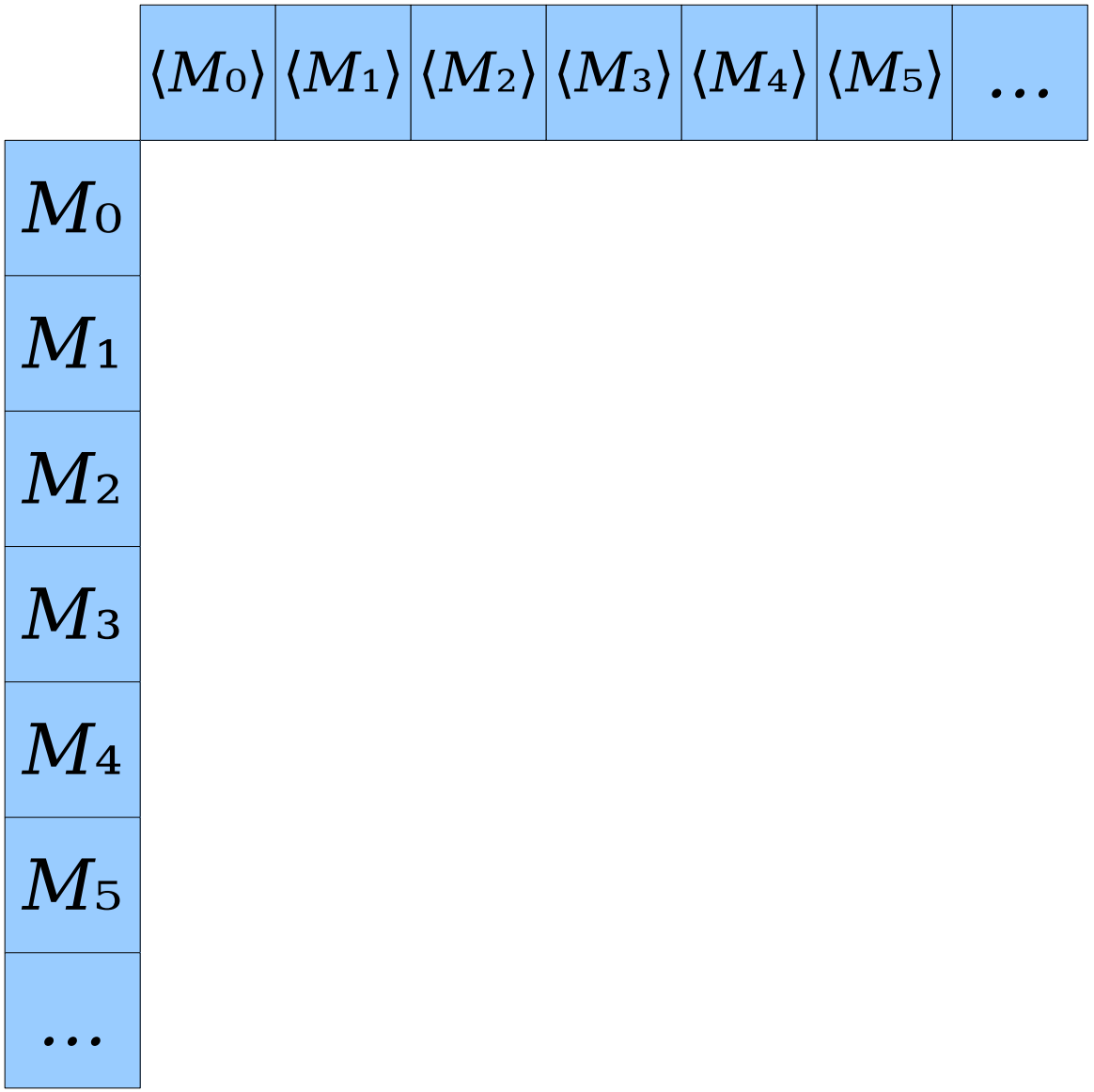
$$\mathscr{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

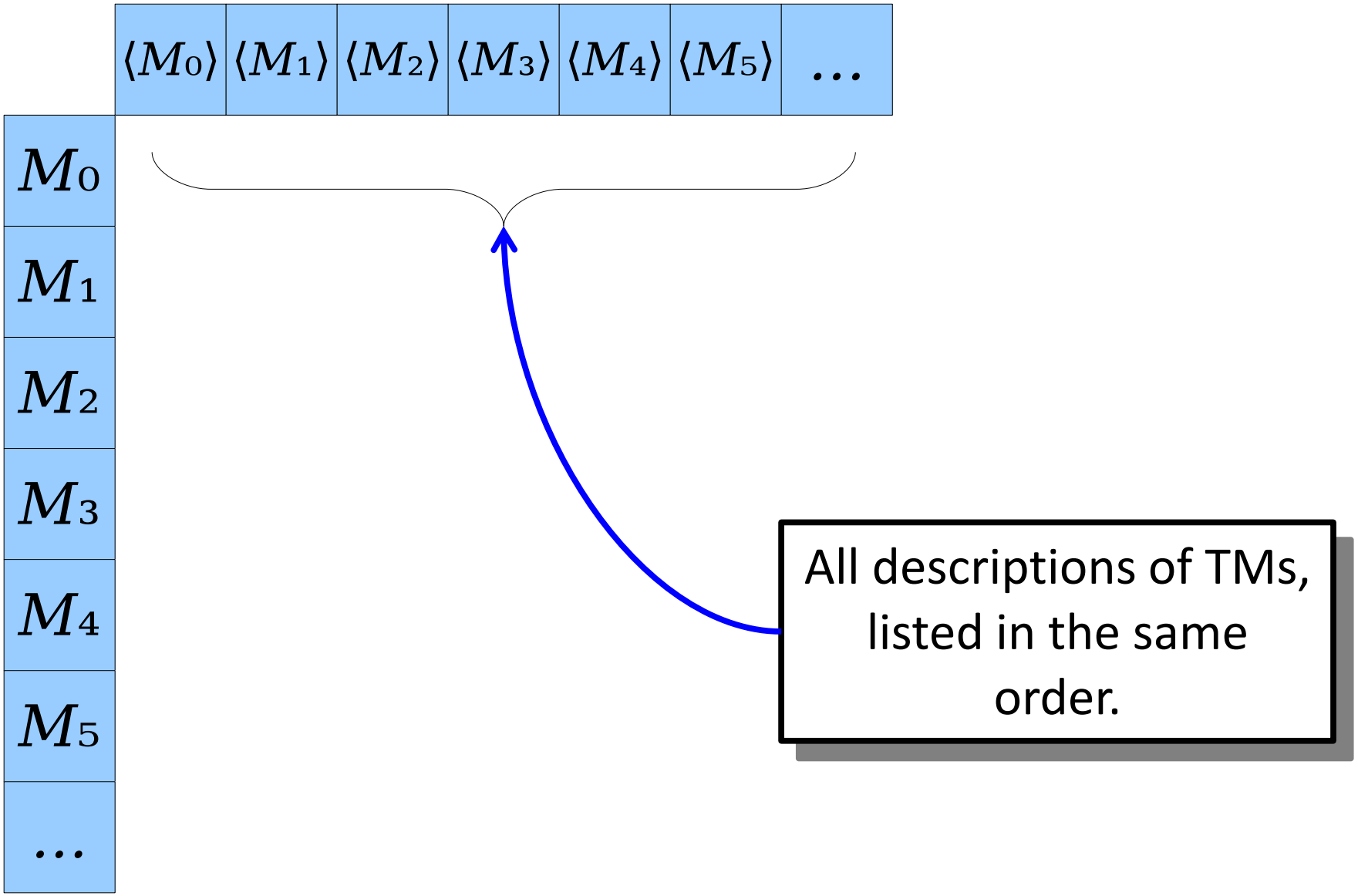
Some of the strings in this set might be descriptions of TMs.

What happens if we list off all Turing machines, looking at how those TMs behave given other TMs as input?


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
...


⟨M0⟩ ⟨M1⟩ ⟨M2⟩ ⟨M3⟩ ⟨M4⟩ ⟨M5⟩ ...
M0
M1
M2
M3
M4
M5
...
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							
...							

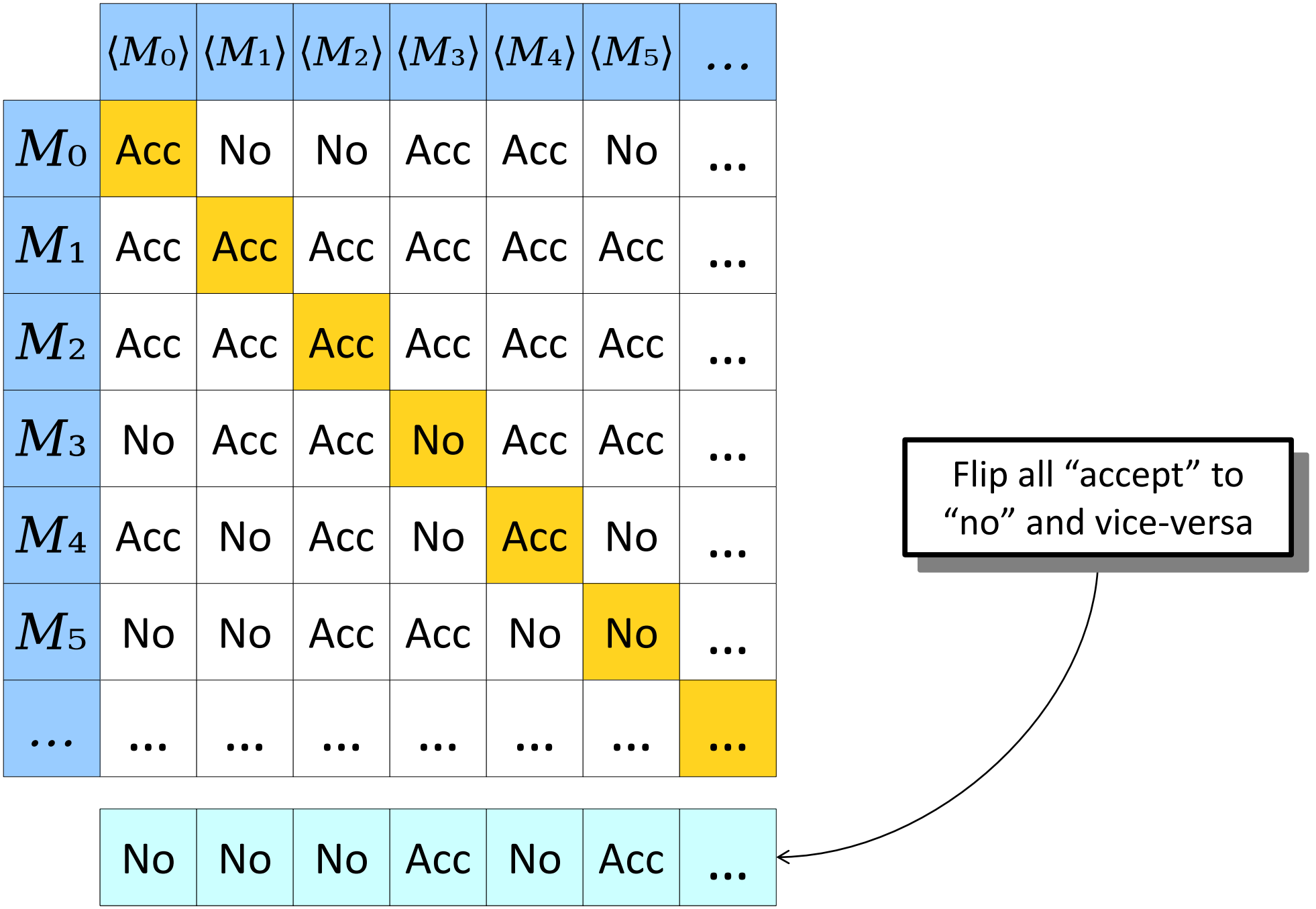
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	...
M_0	Acc	No	No	Acc	Acc	No	...
M_1	Acc	Acc	Acc	Acc	Acc	Acc	...
M_2	Acc	Acc	Acc	Acc	Acc	Acc	...
M_3	No	Acc	Acc	No	Acc	Acc	...
M_4	Acc	No	Acc	No	Acc	No	...
M_5	No	No	Acc	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	...
...	...	...	...	...	...	...	...

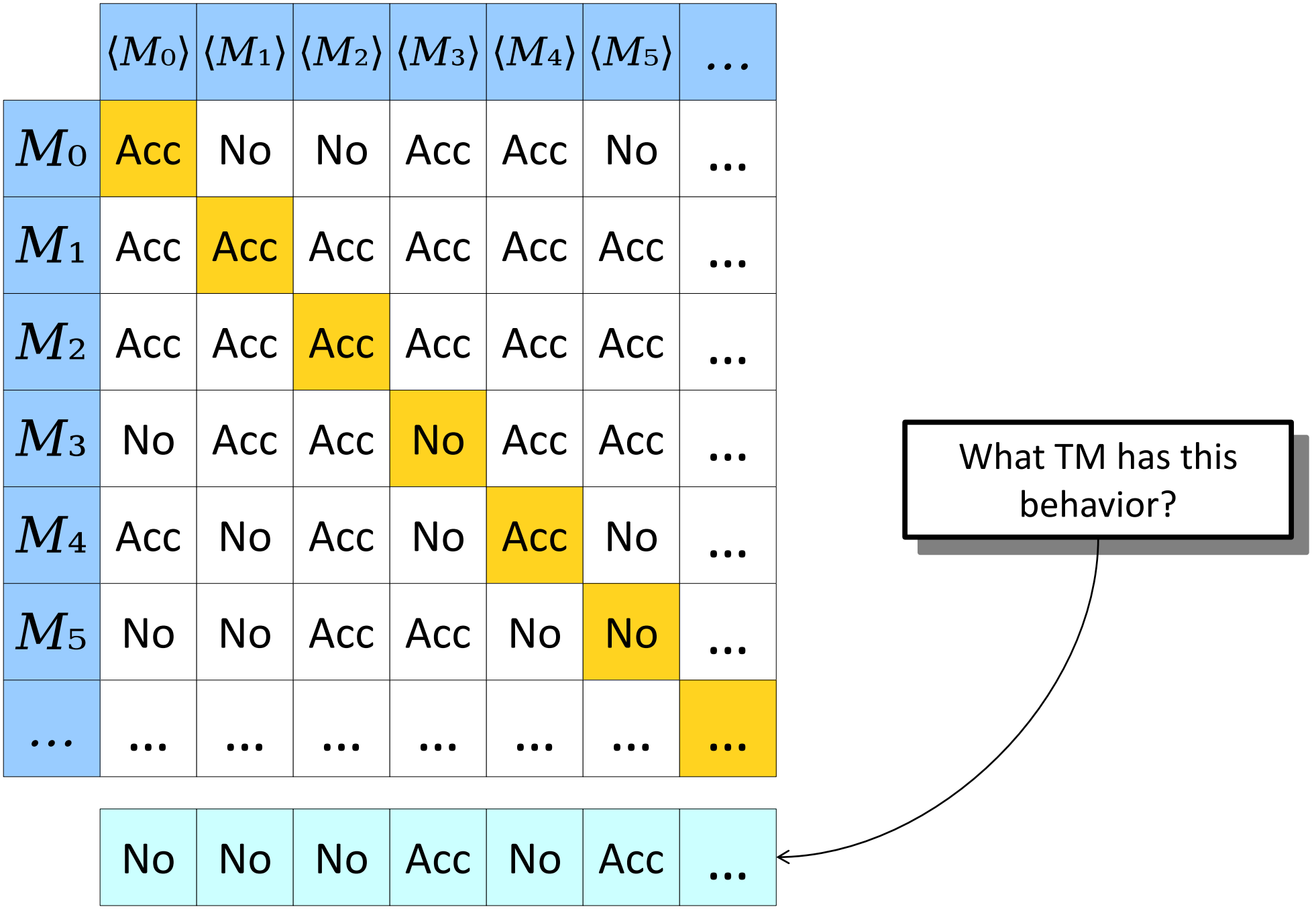
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	...
M_0	Acc	No	No	Acc	Acc	No	...
M_1	Acc	Acc	Acc	Acc	Acc	Acc	...
M_2	Acc	Acc	Acc	Acc	Acc	Acc	...
M_3	No	Acc	Acc	No	Acc	Acc	...
M_4	Acc	No	Acc	No	Acc	No	...
M_5	No	No	Acc	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 ...
... ...
No No No Acc No Acc ...
Flip all “accept” to “no” and vice-versa

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

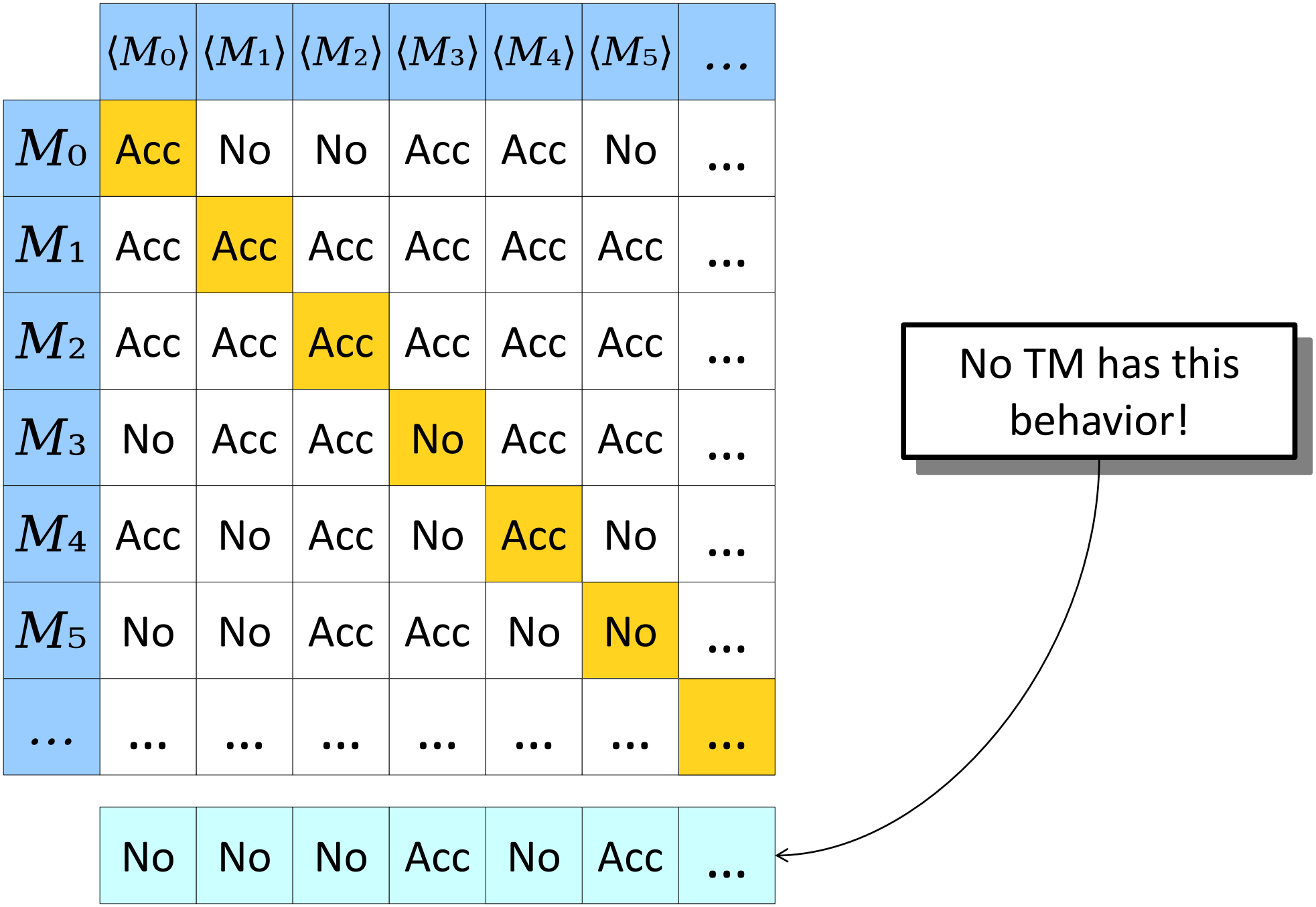
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	...
M_0	Acc	No	No	Acc	Acc	No	...
M_1	Acc	Acc	Acc	Acc	Acc	Acc	...
M_2	Acc	Acc	Acc	Acc	Acc	Acc	...
M_3	No	Acc	Acc	No	Acc	Acc	...
M_4	Acc	No	Acc	No	Acc	No	...
M_5	No	No	Acc	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 descriptions.”

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

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

Diagonalization Revisited

The ***diagonalization language***, which we denote $\boldsymbol{L_D}$, is defined as

$$\boldsymbol{L_D = \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ does not accept } \langle M \rangle \}}$$

We constructed this language to be different from the language of every TM.

Therefore, $L_D \notin \mathbf{RE}$! Let's go prove this.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ does not accept } \langle M \rangle \}$

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

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. This means that there is a TM R where $\mathscr{L}(R) = L_D$.

Now, what happens when we run R on $\langle R \rangle$? We know that

$$R \text{ accepts } \langle R \rangle \quad \text{if and only if} \quad \langle R \rangle \in \mathscr{L}(R).$$

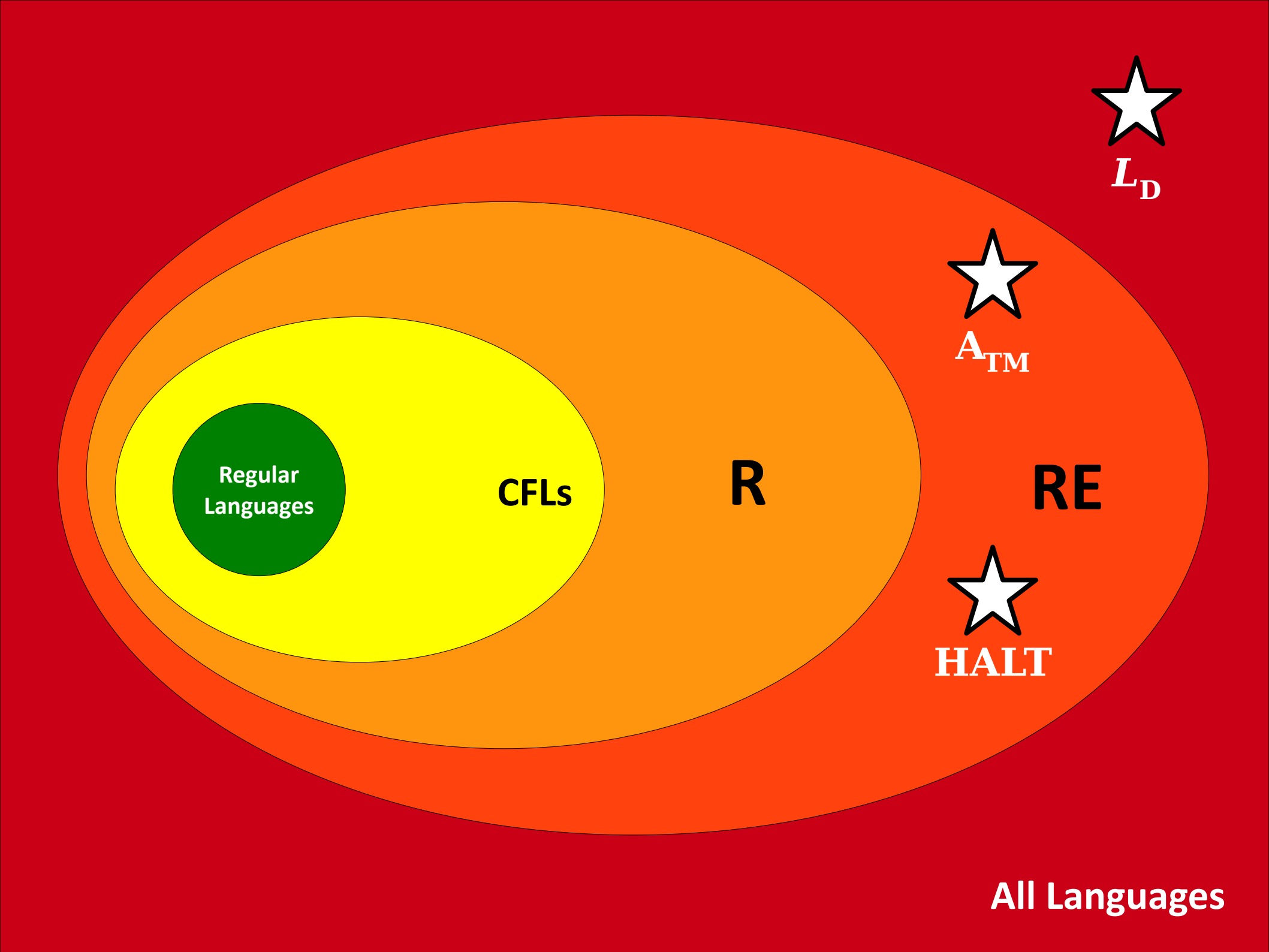
Since $\mathscr{L}(R) = L_D$, the above expression simplifies to

$$R \text{ accepts } \langle R \rangle \quad \text{if and only if} \quad \langle R \rangle \in L_D.$$

Finally, by definition of L_D, we know that $\langle R \rangle \in L_D$ if and only if R does not accept $\langle R \rangle$. Therefore, we see that

$$R \text{ accepts } \langle R \rangle \quad \text{if and only if} \quad R \text{ doesn't accept } \langle R \rangle.$$

This is impossible. We've reached a contradiction, so our assumption was wrong, and so $L_D \notin \mathbf{RE}$. ■


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.

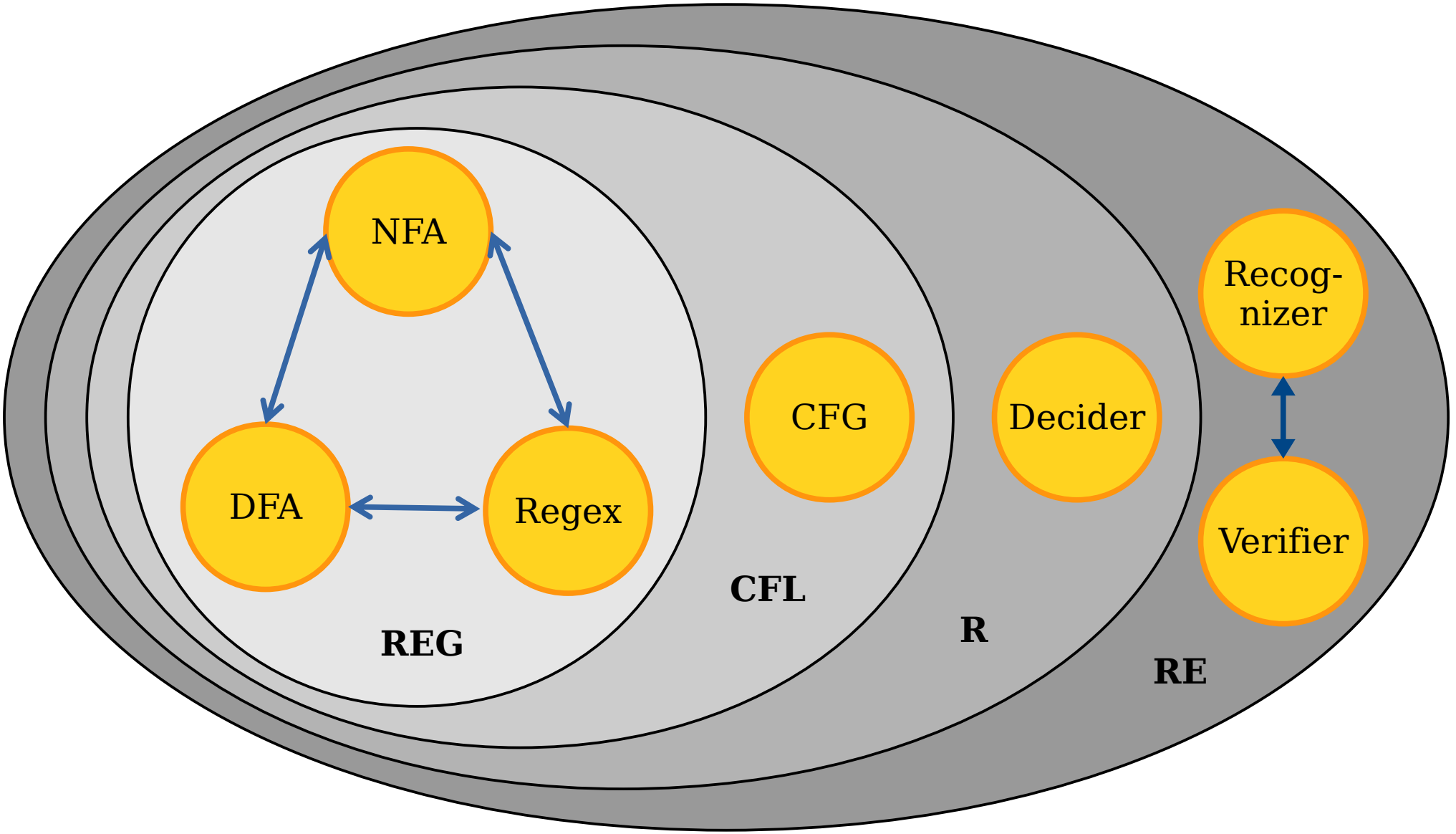
There's no automatic way to do math. There are true statements that we can't prove.

That doesn't mean that mathematics is worthless. It just means that we need to temper our expectations about it.

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!

This is the end of the content we'll be testing you on for the final exam!

The next two lectures on Complexity Theory are purely for your own interest.

Thought for the Weekend:

If it is true, I want to believe it is true.
If it is not true, I want to believe it is not true.