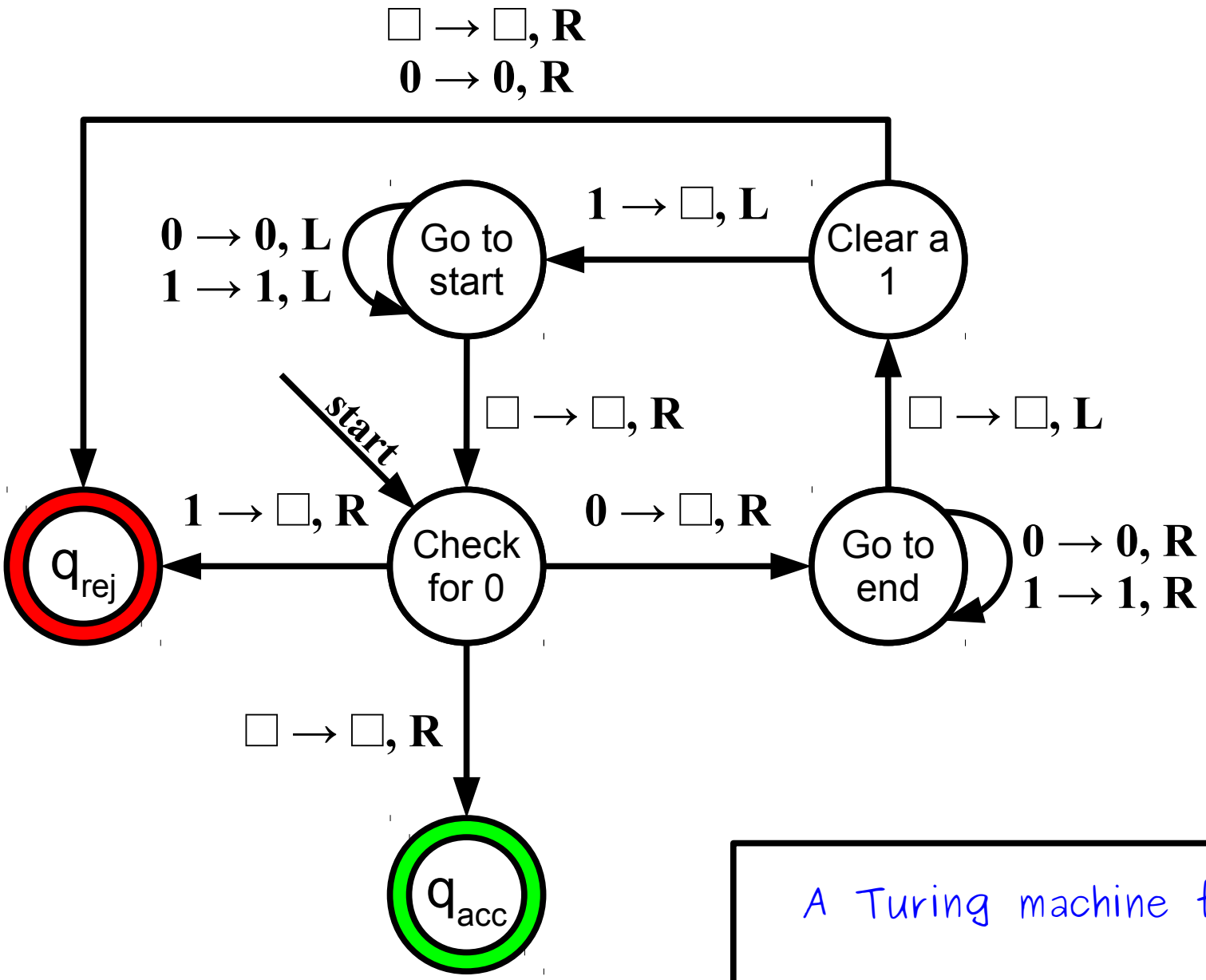
Turing Machines

Part Two? Part Three?

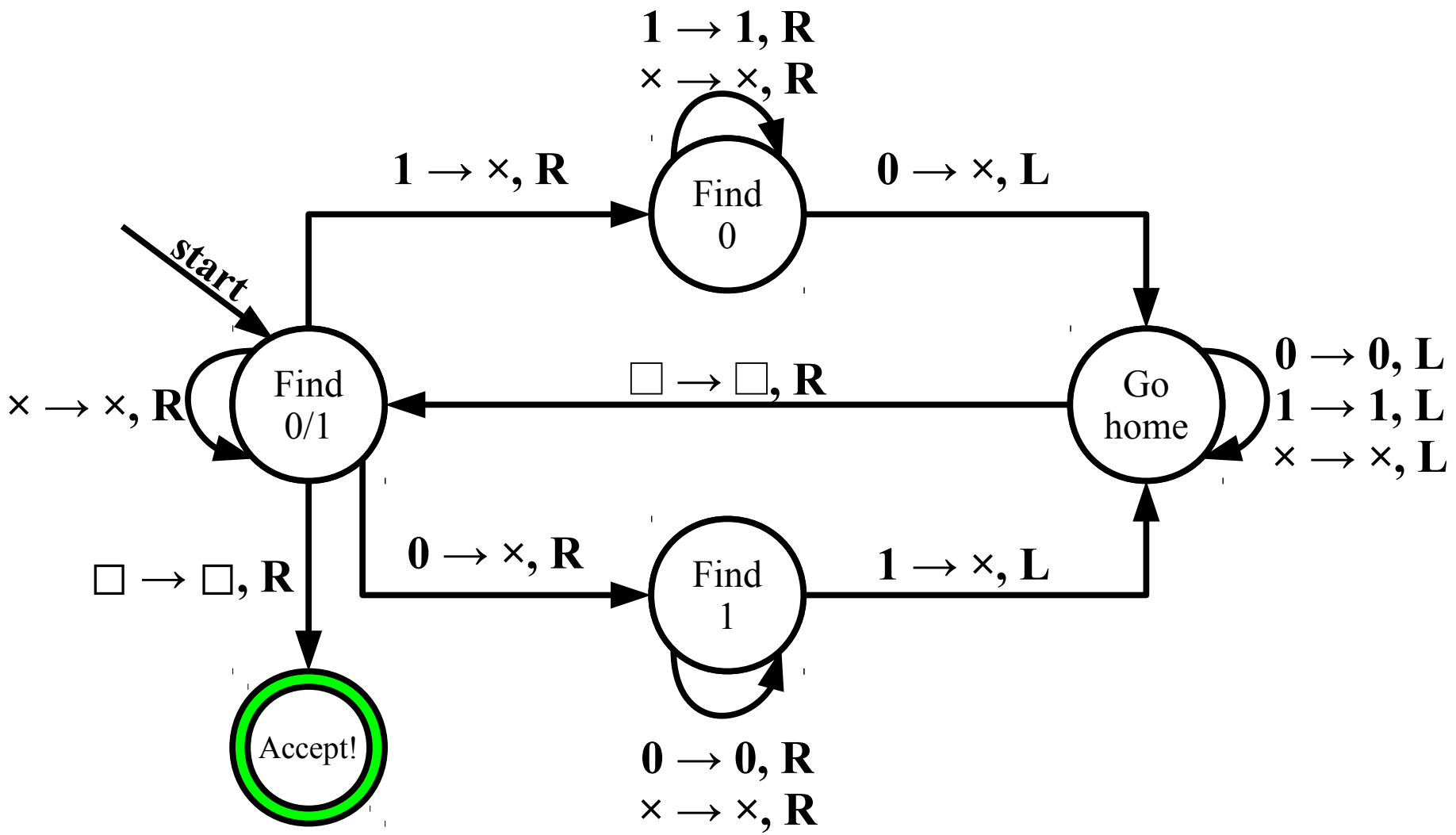
Buckle your seatbelts.
We’ve got a lot to cover today!

Recap from Last Time



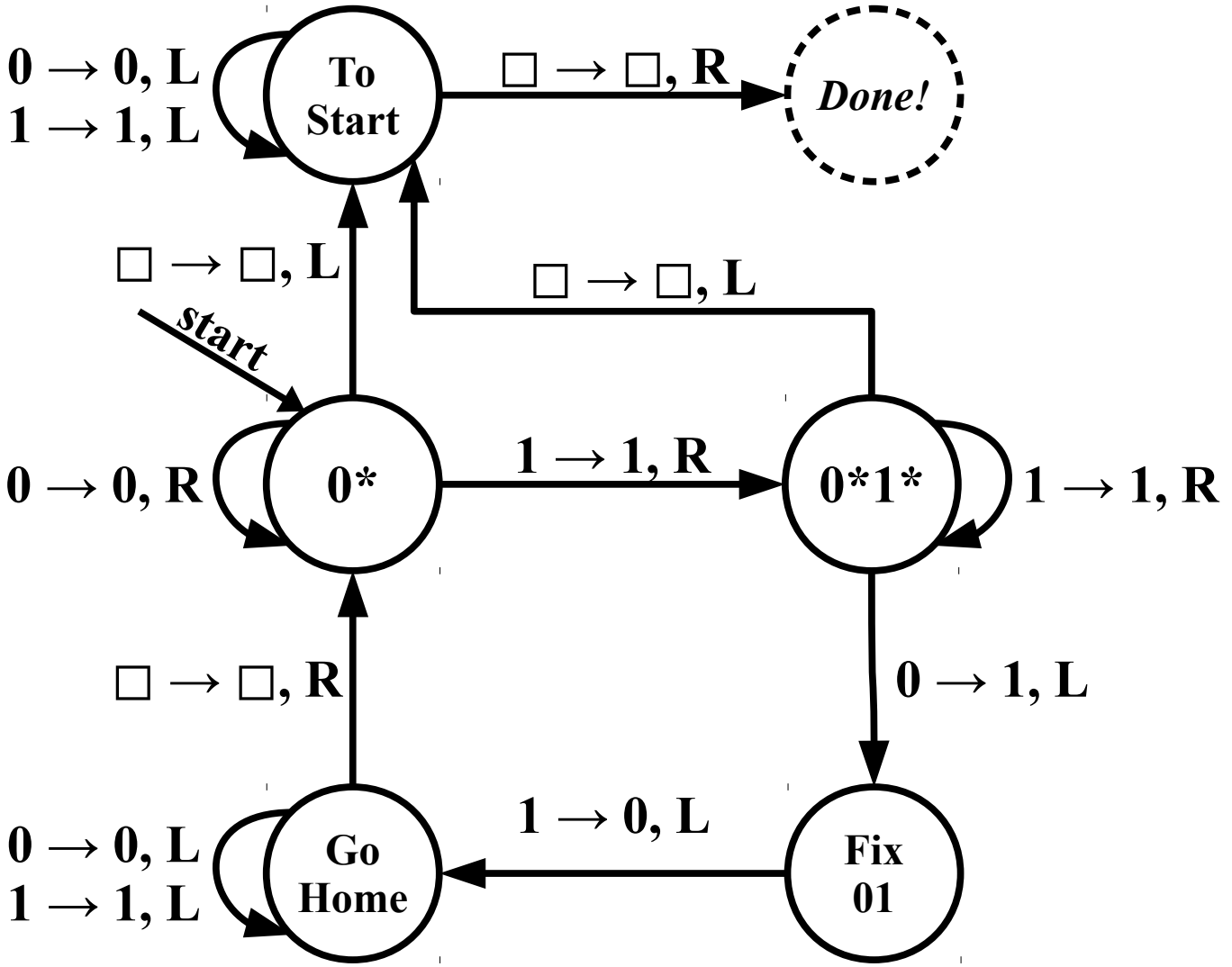
A Turing machine for

$\{ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N} \}$



A Turing machine for

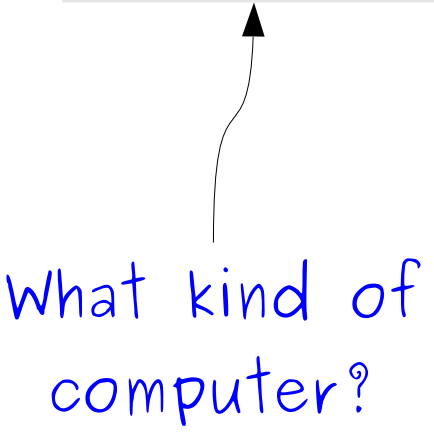
{ $w \in \{0, 1\}^*$ | w has the same number of 0s and 1s }



A TM subroutine that sorts all the 0s and 1s in a string.

New Stuff!

What problems can we solve with a computer?



Where We Stand

- What have we seen TMs do so far?
  - Operate on numbers.
  - Sort sequences of values.
  - Break tasks down into smaller pieces.
- Because last Friday's lecture was canceled, we missed out on a few other tasks TMs can do:
  - Work with base-10 numbers.
  - Increment and decrement numbers.
  - Add numbers.
- Aren't these, you know, the things computers do?

If you're curious to see how this is done, check the website for the lecture slides we would have used. You aren't required to do this, though. It's just a great thing to look into if you're curious!

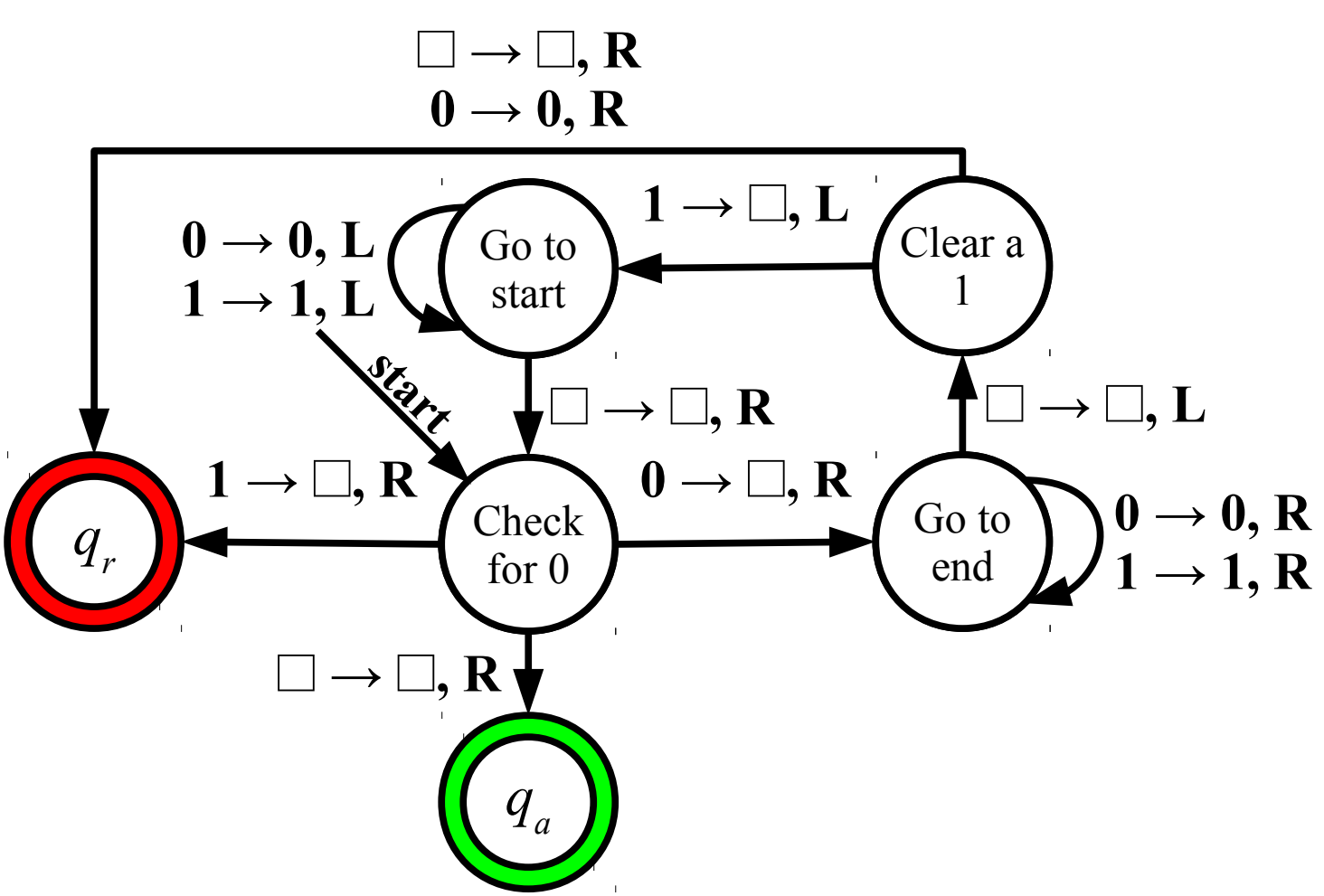
How to Turing machines compare with standard, run-of-the-mill computers?

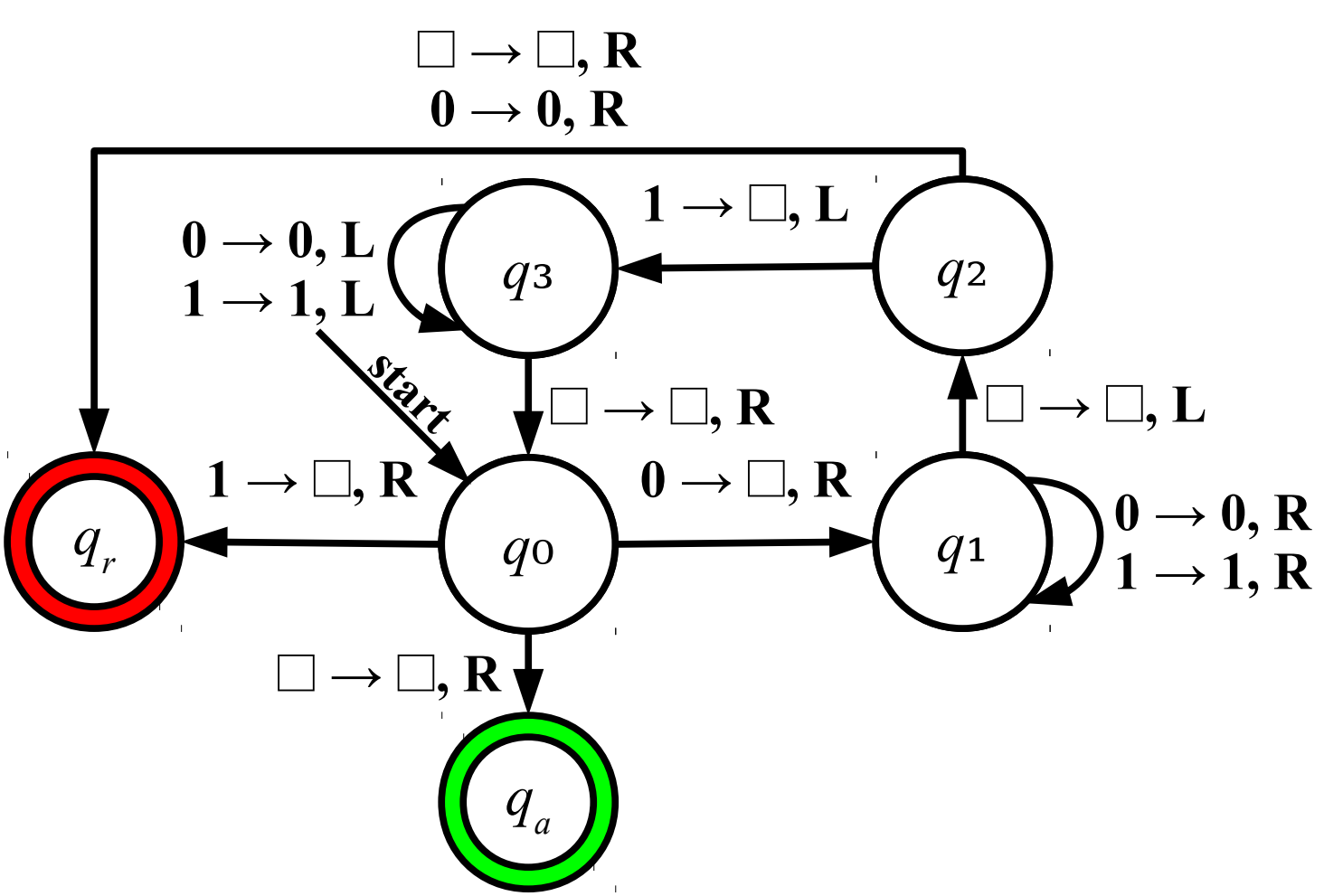
Real and “Ideal” Computers

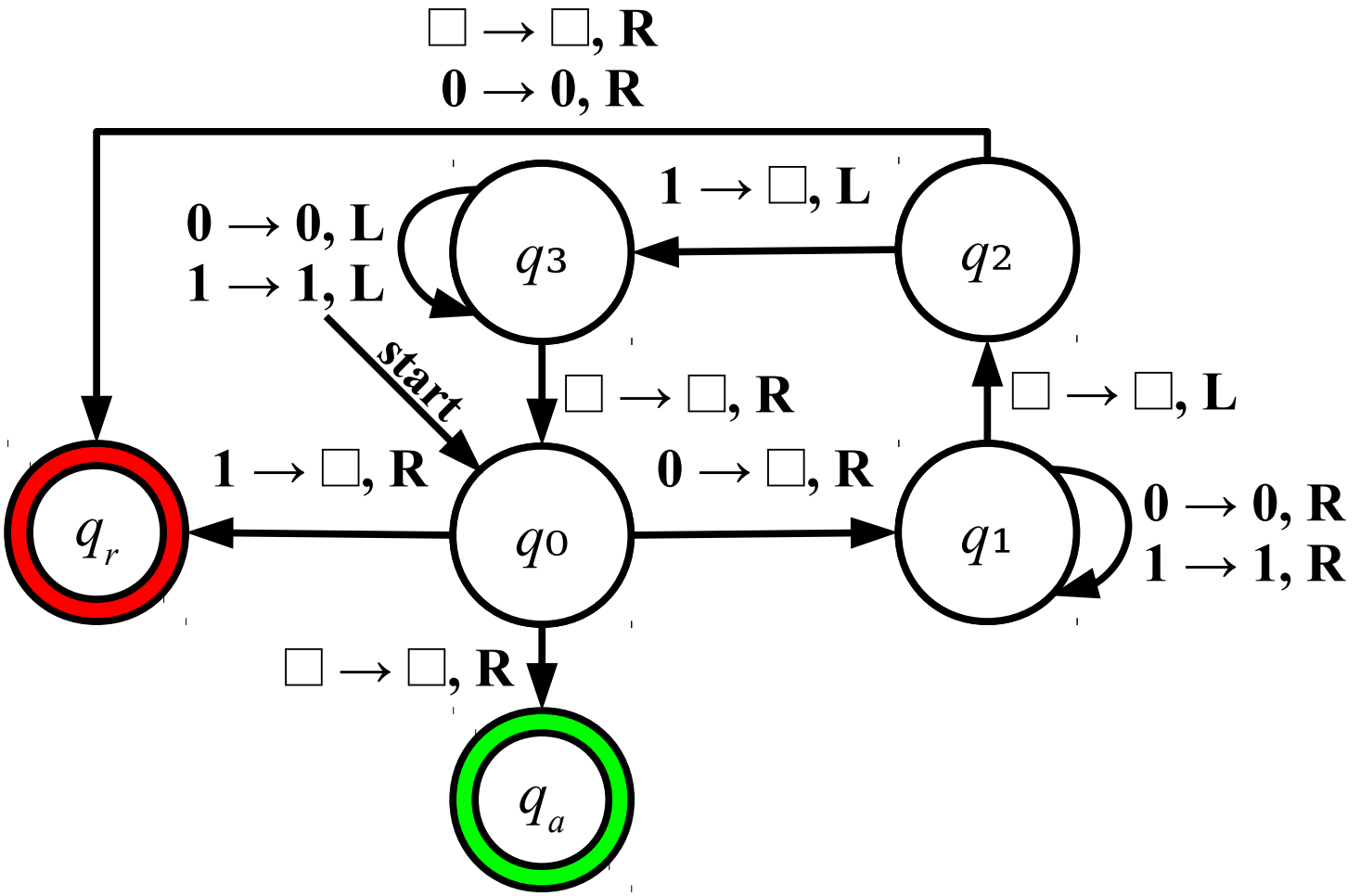
- A real computer has memory limitations: you have a finite amount of RAM, a finite amount of disk space, etc.
  - This makes them equivalent to finite automata.
- However, as computers get more and more powerful, the amount of memory available keeps increasing.
- An ***idealized computer*** is like a regular computer, but with unlimited RAM and disk space. It functions just like a regular computer, but never runs out of memory.

Claim 1: Idealized computers can simulate Turing machines.

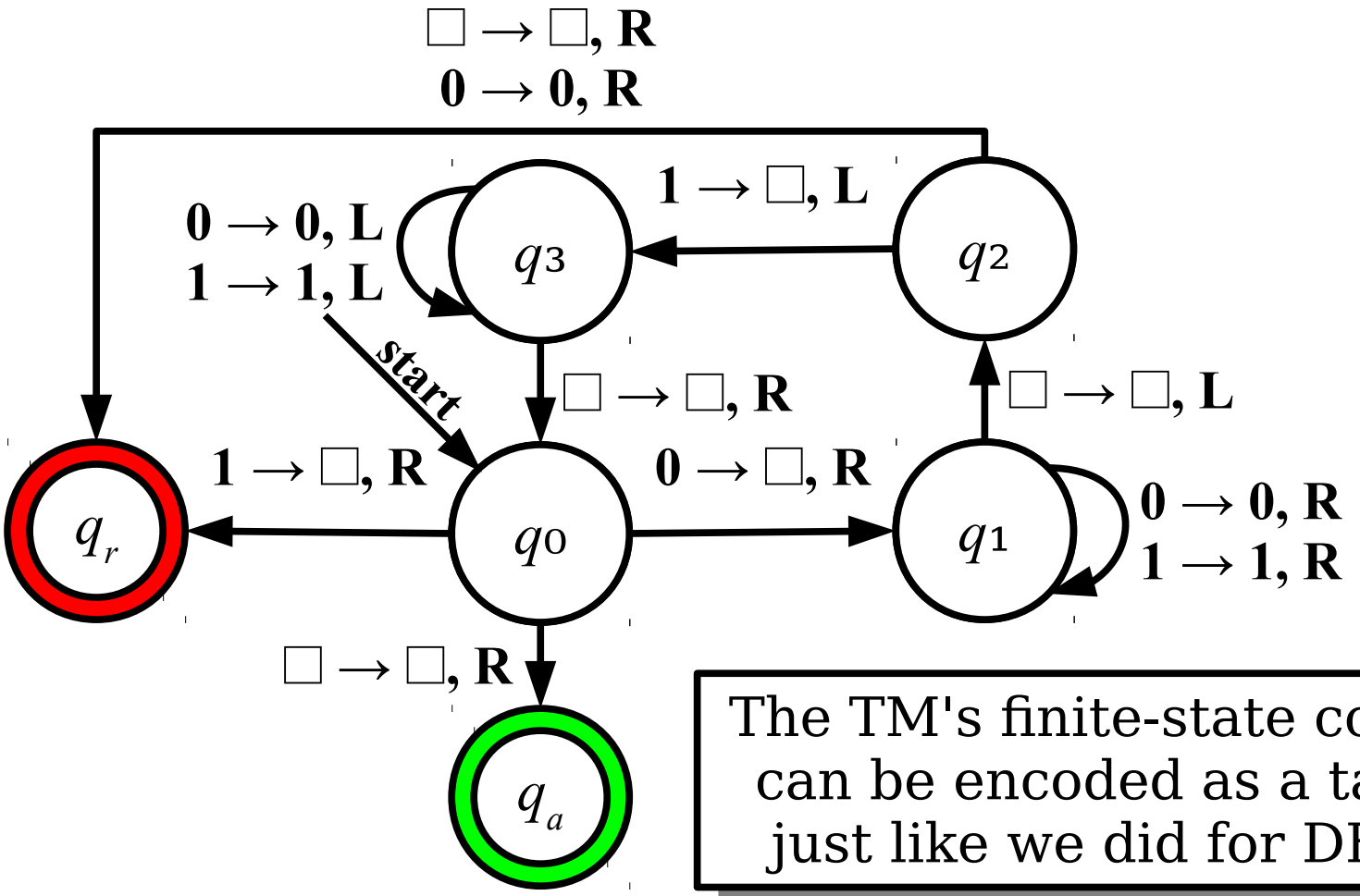
"Anything that can be done with a TM can also be done with an unbounded-memory computer."


□ → □, R
0 → 0, R
0 → 0, L
1 → 1, L
Go to start
1 → □, L
Clear a 1
start
□ → □, R
□ → □, L
1 → □, R
0 → □, R
q_r
Check for 0
Go to end
0 → 0, R
1 → 1, R
□ → □, R
q_a


□ → □, R
0 → 0, R
1 → □, L
0 → 0, L
1 → 1, L
q3
q2
start
□ → □, R
□ → □, L
1 → □, R
0 → □, R
0 → 0, R
1 → 1, R
qr
q0
q1
□ → □, R
qa



	0			1			□		
q_0	q_1	□	R	q_r	□	R	q_a	□	R
q_1	q_1	0	R	q_1	1	R	q_2	□	L
q_2	q_r	0	R	q_3	□	L	q_r	□	R
q_3	q_3	0	L	q_3	1	L	q_0	□	R

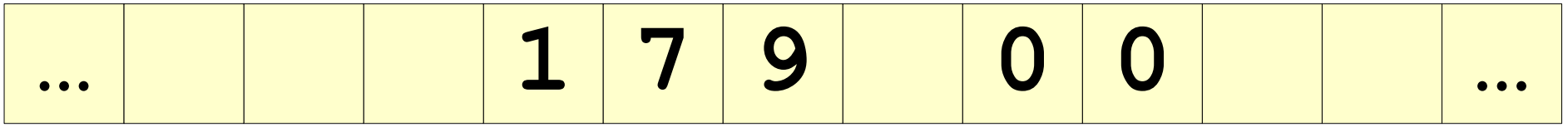


The TM's finite-state control can be encoded as a table, just like we did for DFAs.

	0			1			□		
q_0	q_1	□	R	q_r	□	R	q_a	□	R
q_1	q_1	0	R	q_1	1	R	q_2	□	L
q_2	q_r	0	R	q_3	□	L	q_r	□	R
q_3	q_3	0	L	q_3	1	L	q_0	□	R

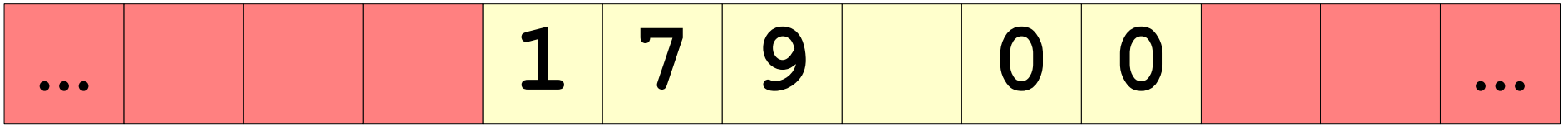
Simulating a TM

- To simulate a TM, the computer would need to be able to keep track of
  - the finite-state control,
  - the current state,
  - the position of the tape head, and
  - the tape contents.
- The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
- We only need to store the “interesting” part of the tape (the parts that have been read from or written to so far.)



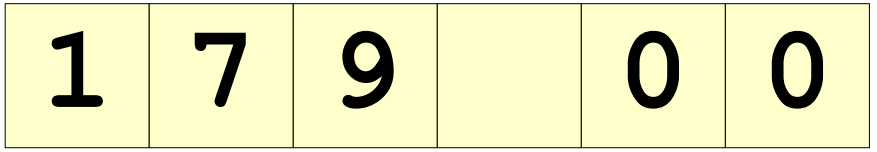
Simulating a TM

- To simulate a TM, the computer would need to be able to keep track of
  - the finite-state control,
  - the current state,
  - the position of the tape head, and
  - the tape contents.
- The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
- We only need to store the “interesting” part of the tape (the parts that have been read from or written to so far.)



Simulating a TM

- To simulate a TM, the computer would need to be able to keep track of
  - the finite-state control,
  - the current state,
  - the position of the tape head, and
  - the tape contents.
- The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
- We only need to store the “interesting” part of the tape (the parts that have been read from or written to so far.)



Claim 2: Turing machines can simulate idealized computers.

"Anything that can be done with an unbounded-memory computer can be done with a TM."

What We've Seen

- TMs can
  - implement loops (basically, every TM we've seen).
  - make function calls (subroutines).
  - keep track of natural numbers (written in unary or in decimal on the tape).
  - perform elementary arithmetic (equality testing, multiplication, addition, increment, decrement, etc.).
  - perform if/else tests (different transitions based on different cases).

What Else Can TMs Do?

- Maintain variables.
  - Have a dedicated part of the tape where the variables are stored.
  - We've seen this before: you can kinda sorta think of our machine for $\{ \texttt{0}^n\texttt{1}^n \mid n \in \mathbb{N} \}$ as checking if two variables are equal.
- Maintain arrays and linked structures.
  - Divide the tape into different regions corresponding to memory locations.
  - Represent arrays and linked structures by keeping track of the ID of one of those regions.

A CS107 Perspective

- Internally, computers execute by using basic operations like
  - simple arithmetic,
  - memory reads and writes,
  - branches and jumps,
  - register operations,
  - etc.
- Each of these are simple enough that they could be simulated by a Turing machine.

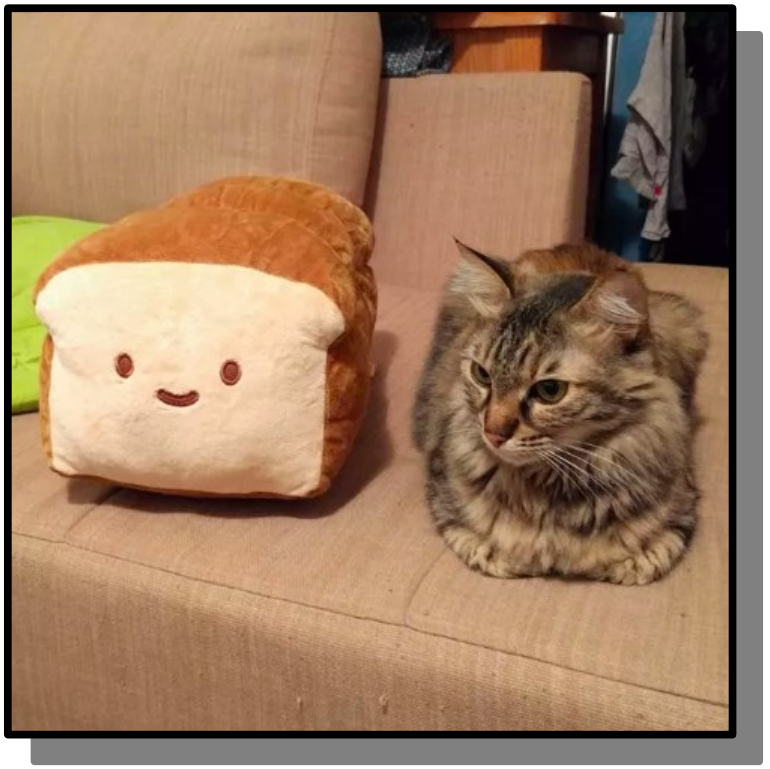
A Leap of Faith

- It may require a leap of faith, but anything you can do with a computer (excluding randomness and user input) can be performed by a Turing machine.
- The resulting TM might be colossal, or really slow, or both, but it would still faithfully simulate the computer.
- We're going to take this as an article of faith in CS103. If you’re curious for more details, come talk to me after class.

Wait, You're Saying a TM Can Do...

"cat pictures?"

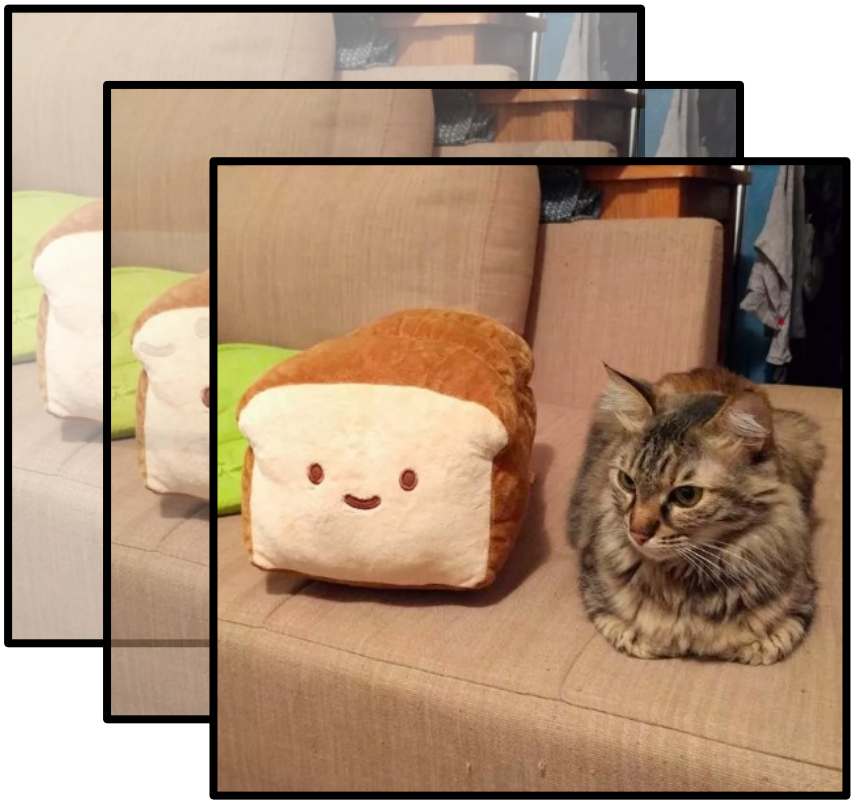
Sure! A picture is just a 2D array of colors, and a color can be represented as a series of numbers.



Wait, You're Saying a TM Can Do...

~~"cat pictures?"~~

"cat videos?"



If you think about it, a video is just a series of pictures!

Wait, You're Saying a TM Can Do...

"music?"

Yes! Write encodings of notes to play on the TM tape. Hook up a speaker device that reads the tape and makes sound.

"chat messages over the internet?"

Yes! View all networked computers as one gigantic machine.

Just how powerful ***are*** Turing machines?

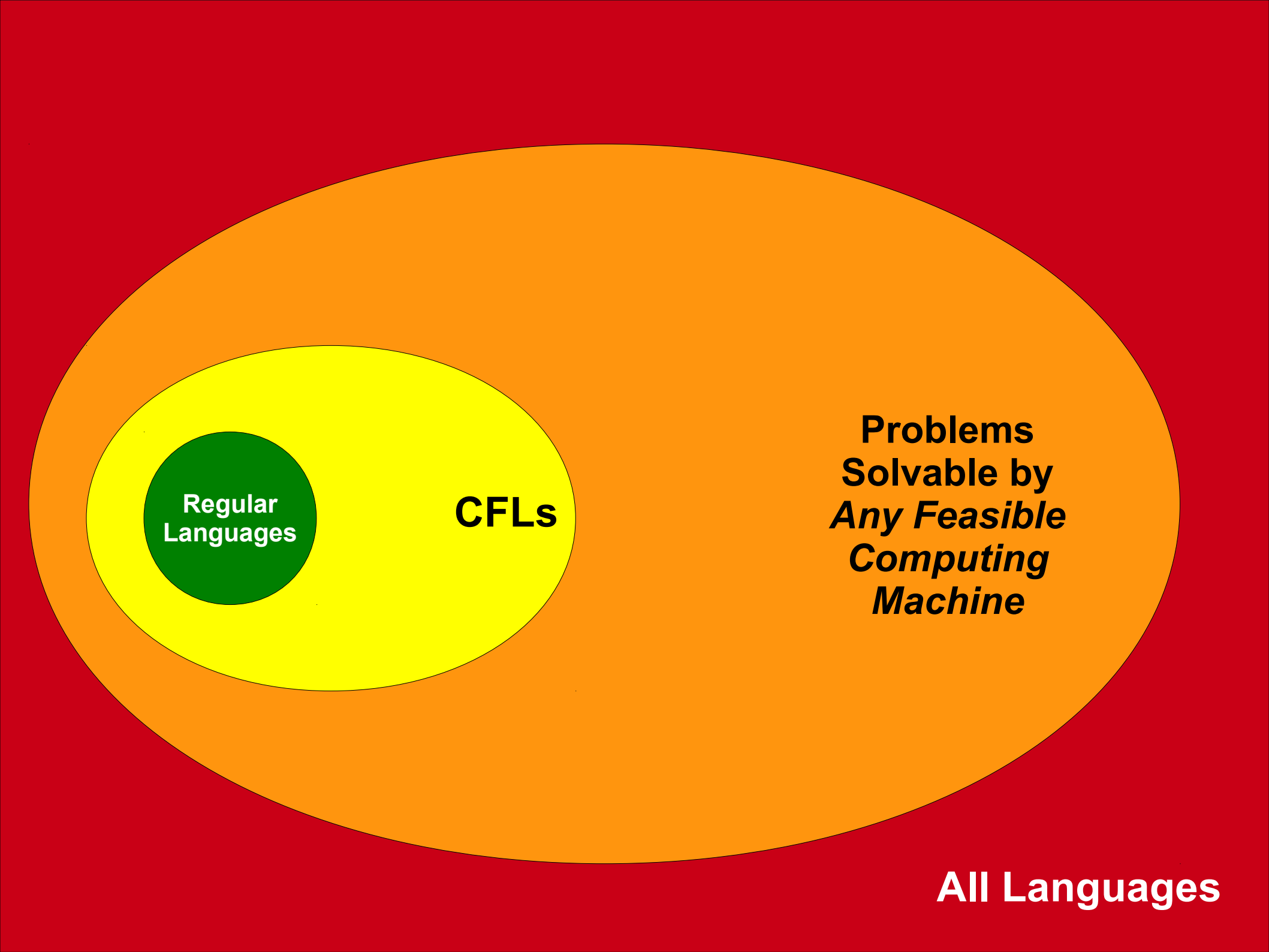
Effective Computation

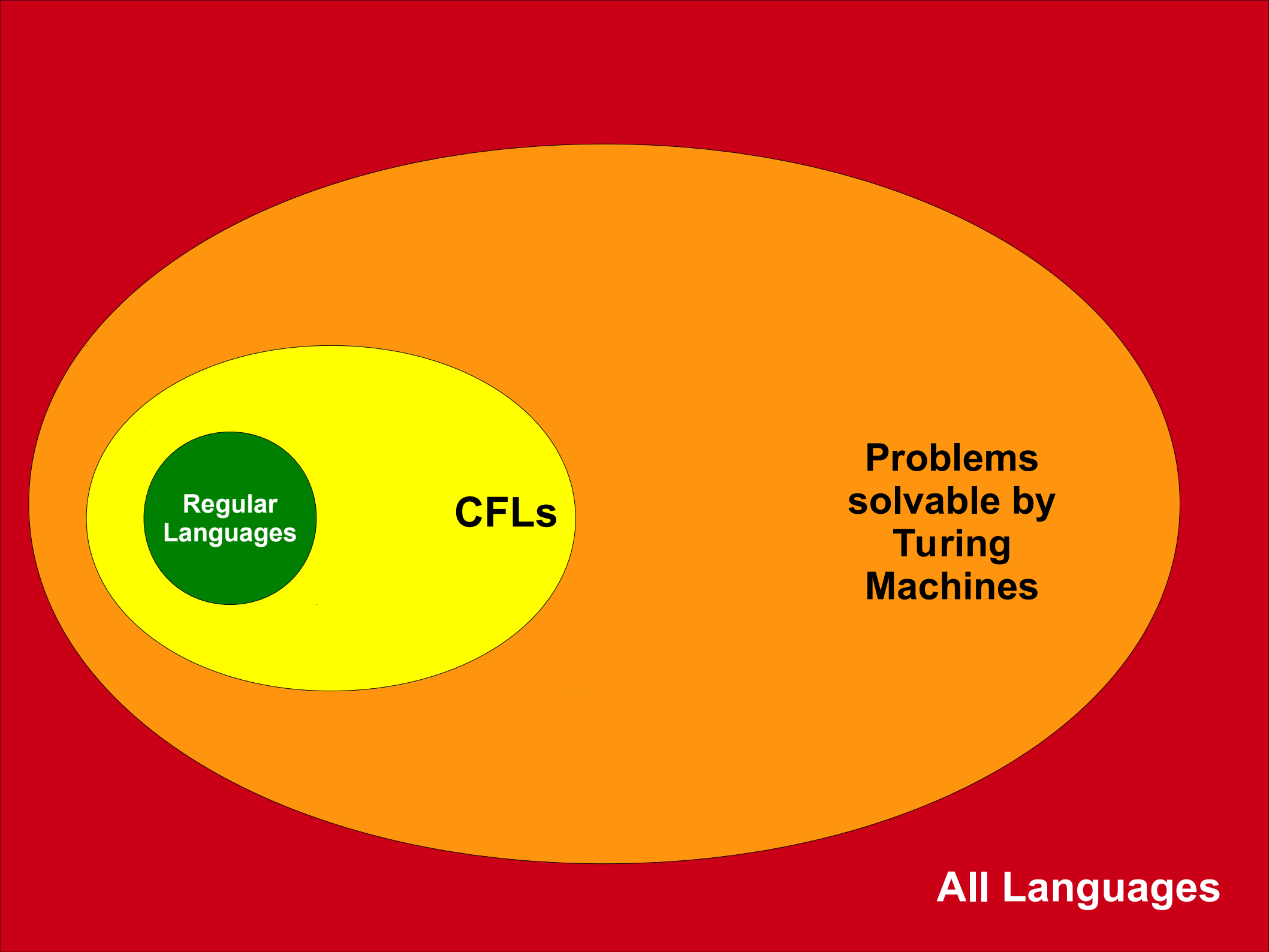
- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
- This is not a formal definition. Rather, it's a set of properties we expect out of a computational system.

The ***Church-Turing Thesis*** claims that

every effective method of computation is either equivalent to or weaker than a Turing machine.

> “This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!”
>
> - Ryan Williams

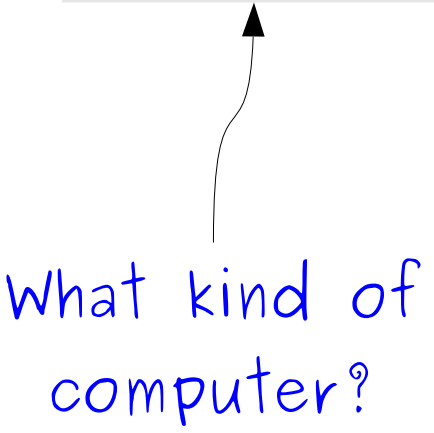

Regular Languages
CFLs
Problems Solvable by *Any Feasible Computing Machine*
All Languages


Regular Languages
CFLs
Problems solvable by Turing Machines
All Languages

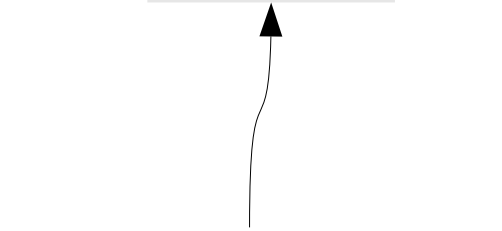
TMs ≈ Computers

- Because Turing machines have the same computational powers as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.
- Going forward, we're going to switch back and forth between TMs and computer programs based on whatever is most appropriate.
- In fact, our eventual proofs about the existence of impossible problems will involve a good amount of pseudocode. Stay tuned for details!

What problems can we solve with a computer?

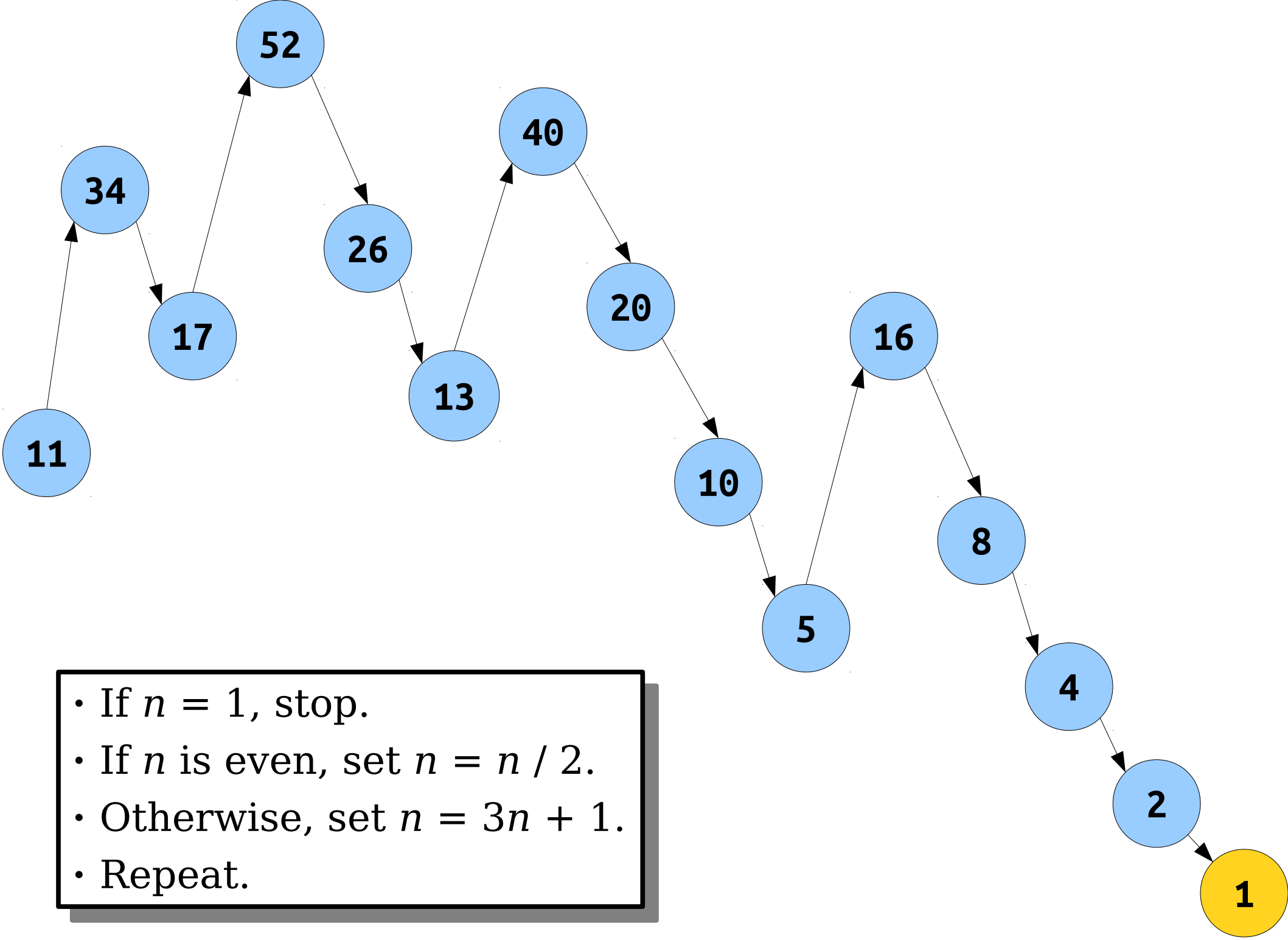


What problems can we solve with a computer?



The Hailstone Sequence

- Consider the following procedure, starting with some $n \in \mathbb{N}$, where $n > 0$:
  - If $n = 1$, you are done.
  - If n is even, set $n = n / 2$.
  - Otherwise, set $n = 3n + 1$.
  - Repeat.
- ***Question:*** Given a number n, does this process terminate?



- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- Otherwise, set $n = 3n + 1$.
- Repeat.

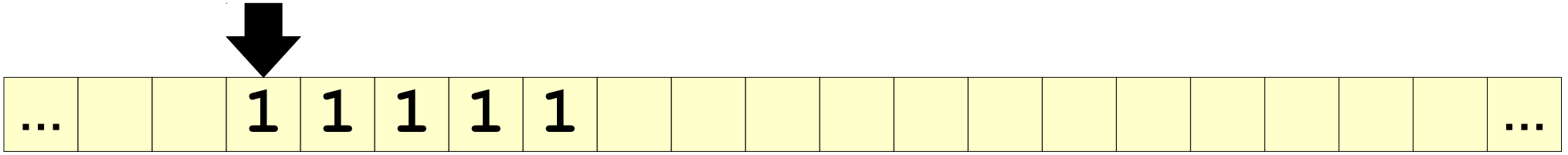
The Hailstone Sequence

- Let $\Sigma = \{\mathtt{1}\}$ and consider the language

  $L = \{ \mathtt{1}^n \mid n > 0$ and the hailstone sequence terminates for $n \}$.

- Could we build a TM for L?

The Hailstone Turing Machine



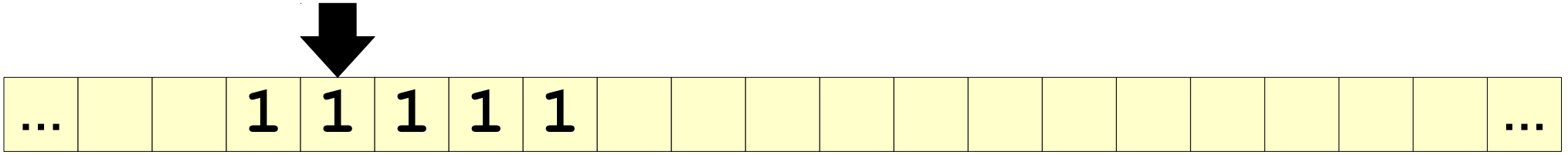
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



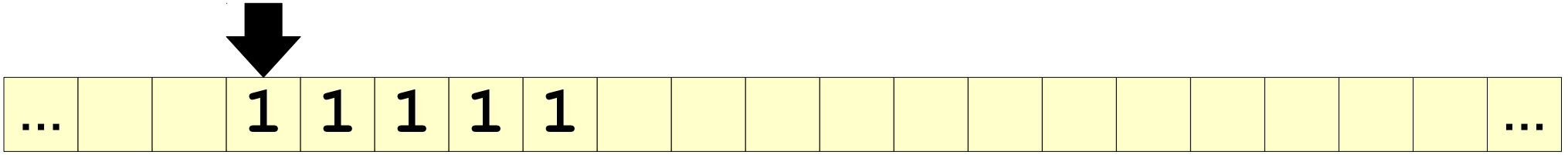
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



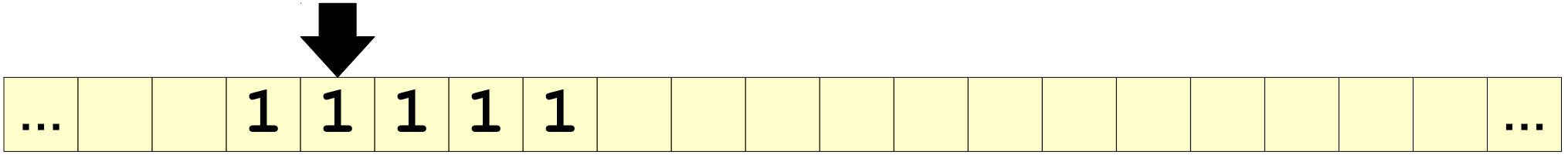
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



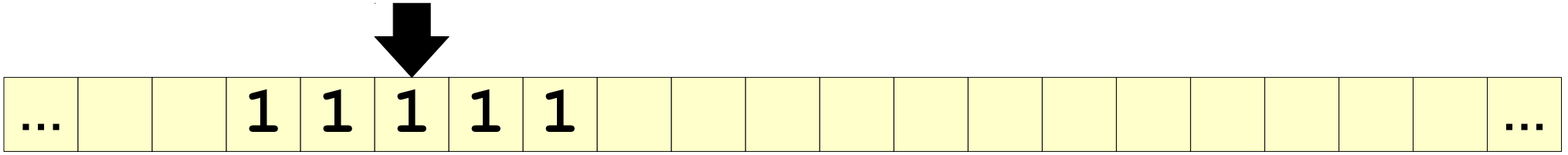
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



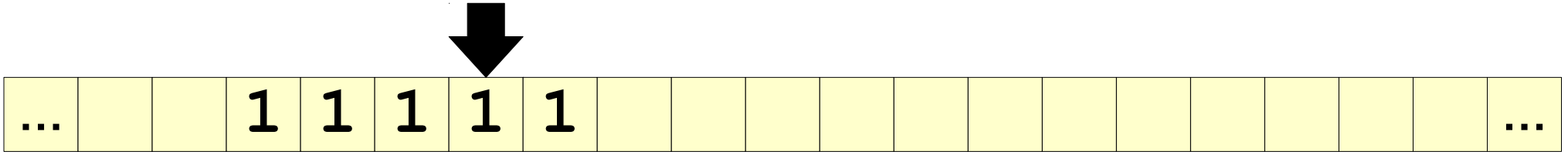
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



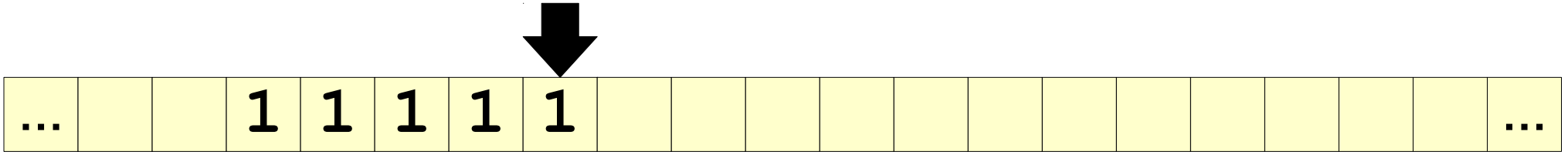
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



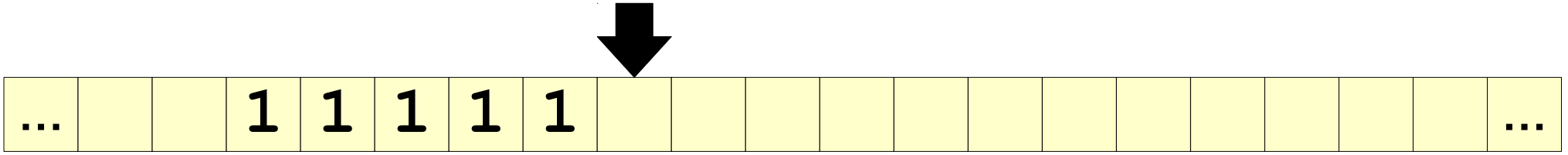
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



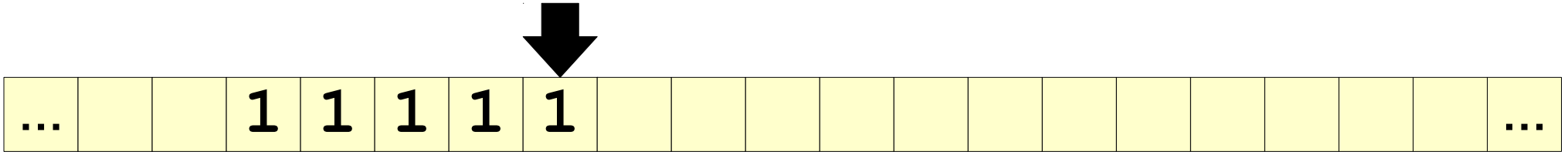
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



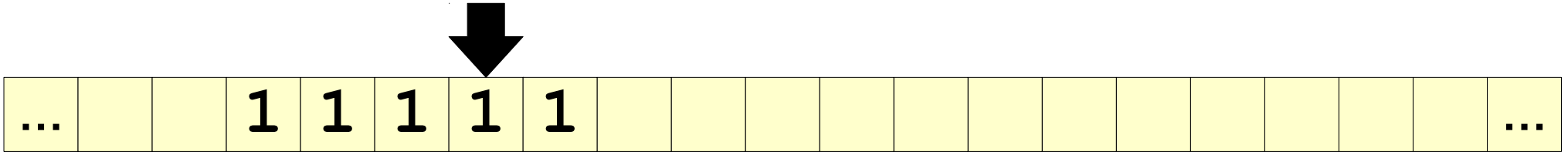
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



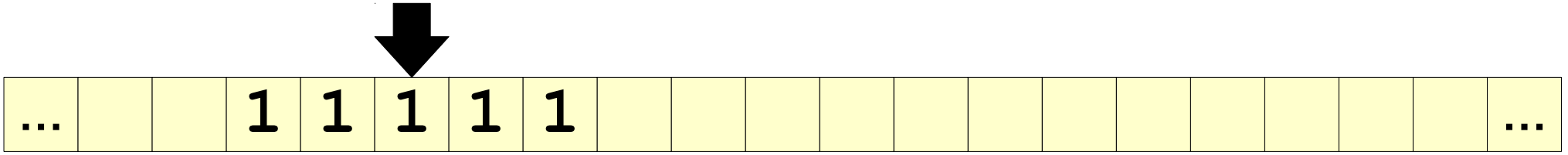
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



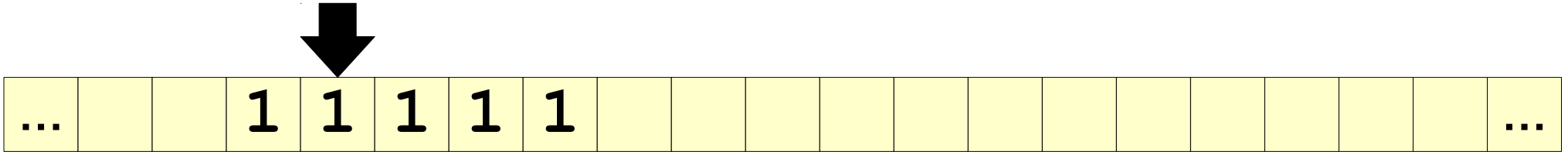
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



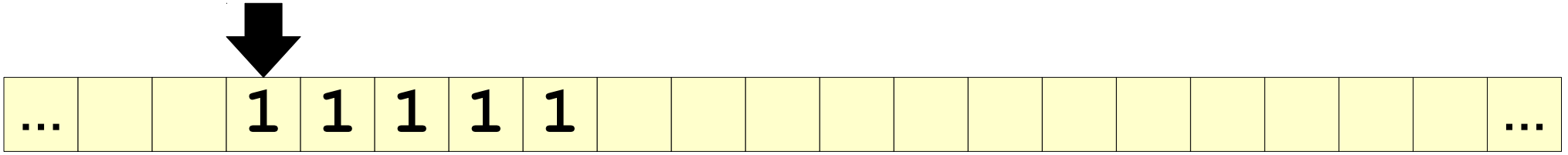
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



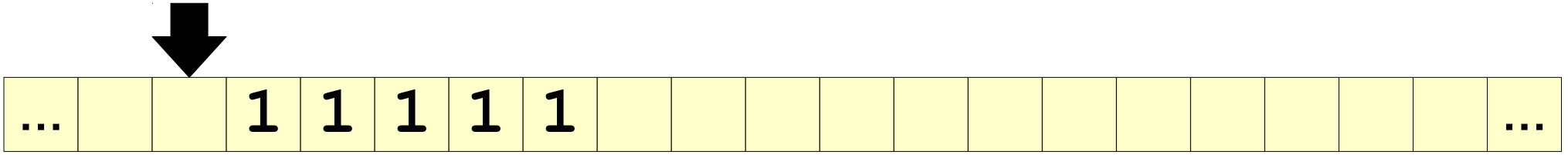
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



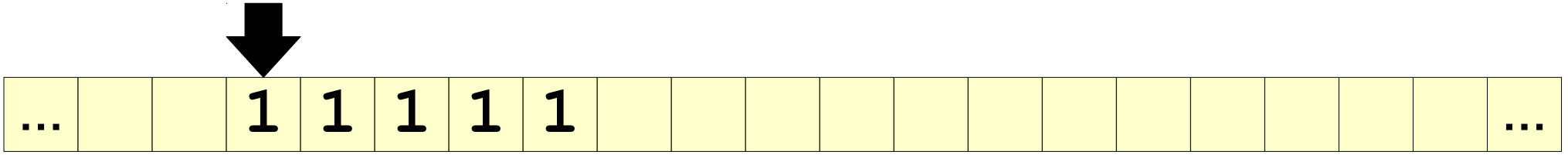
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



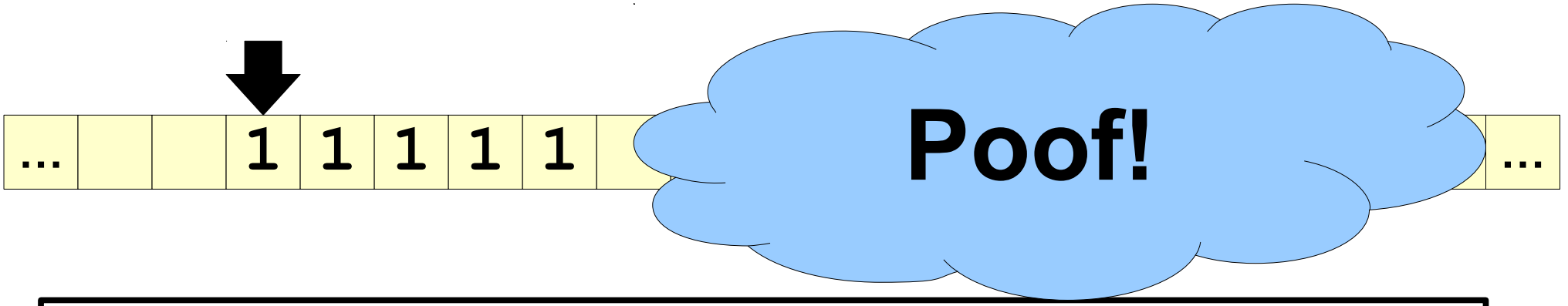
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



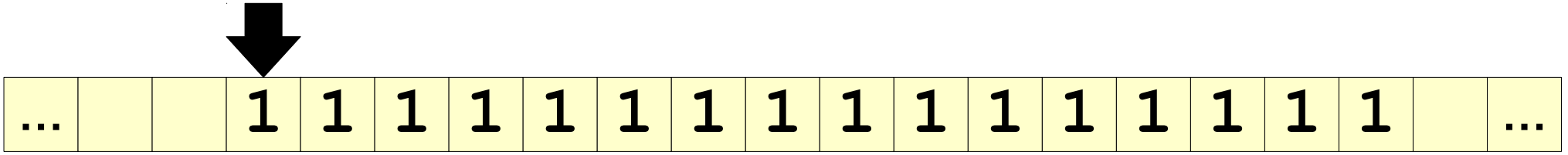
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



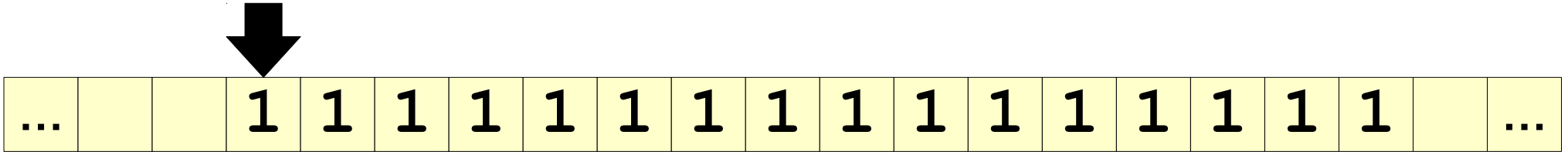
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



If the input is ε, reject.

While the input is not 1:

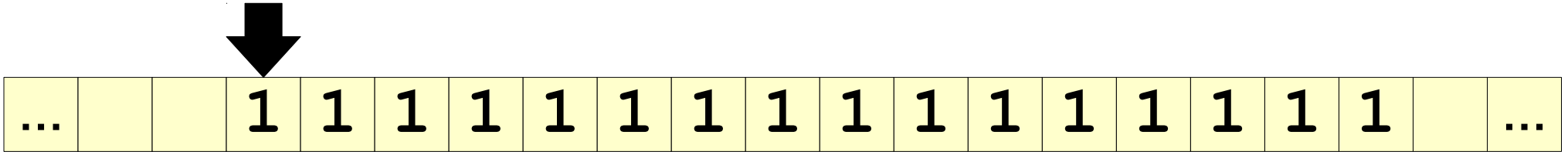
- If the input has even len
  of the string.
- If the input has odd leng
  string and append a 1.

Accept.

Problem Set Question:

Build a TM that, starting with n 1s on its tape, ends with $3n + 1$ 1s on its tape.

The Hailstone Turing Machine



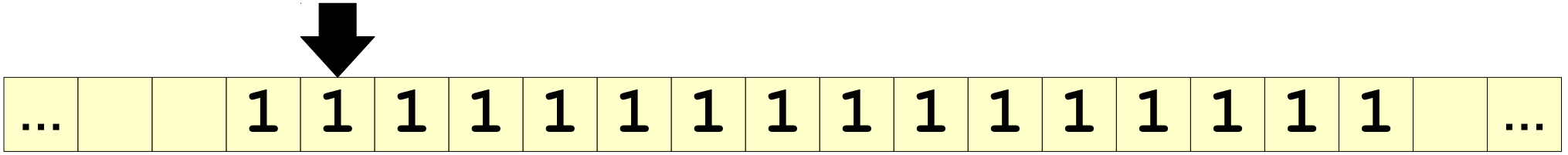
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



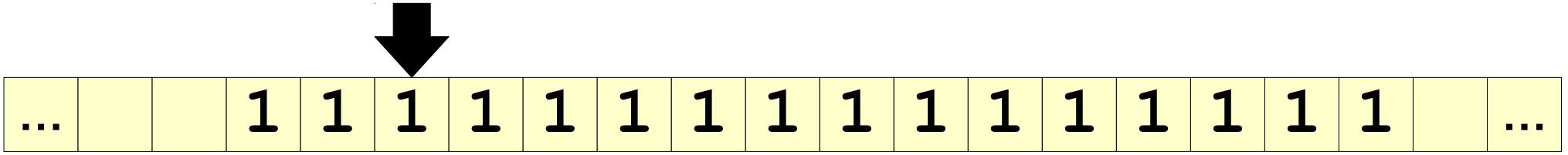
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



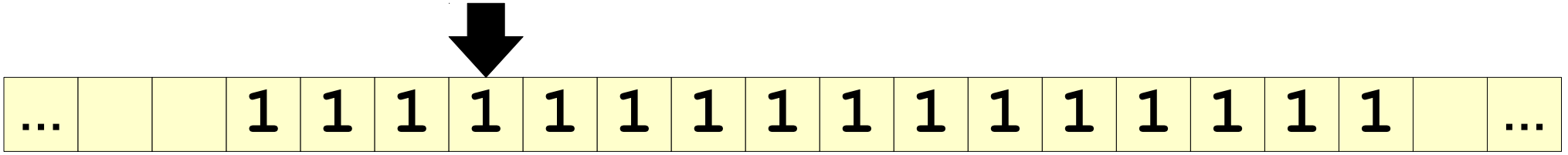
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



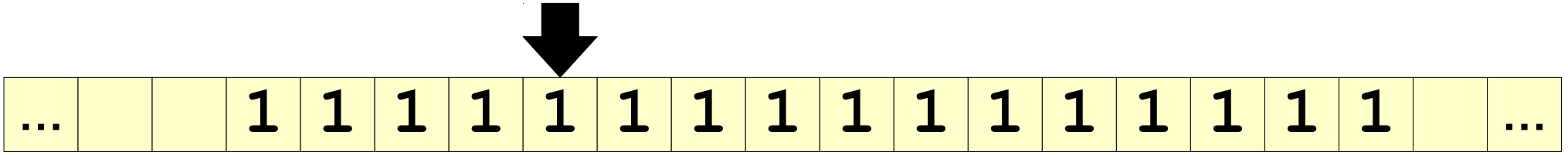
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



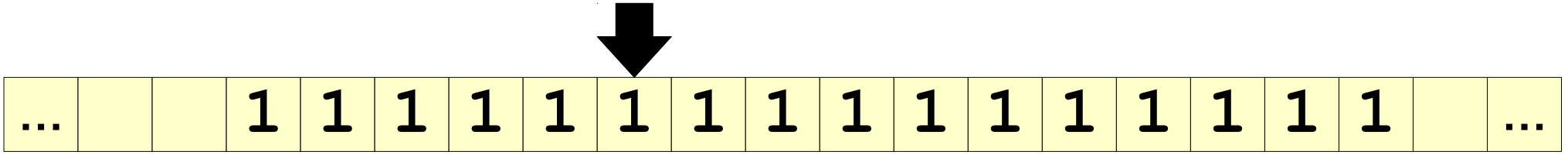
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



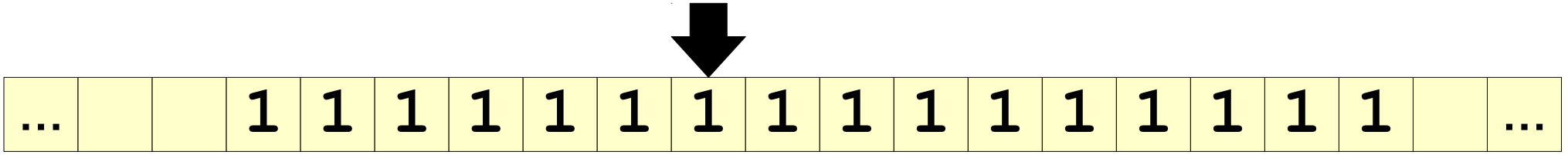
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



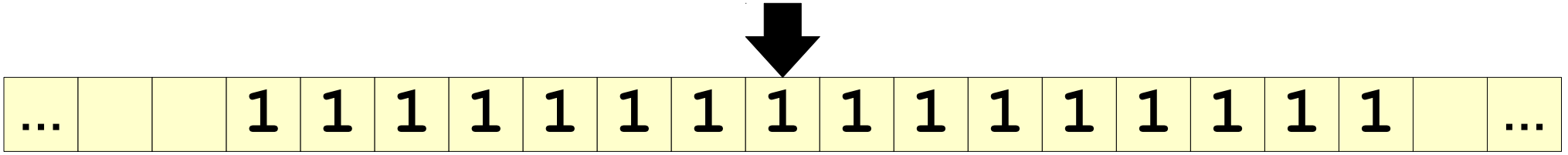
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



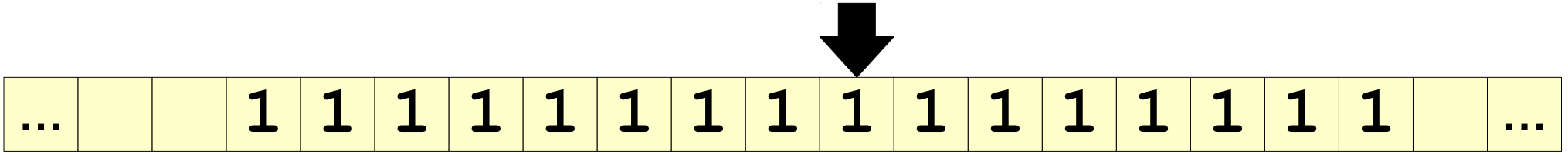
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



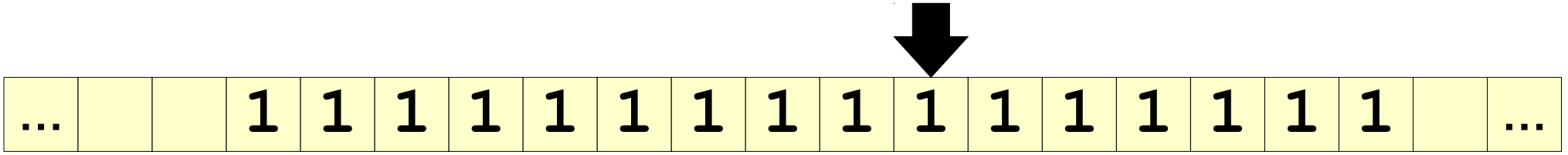
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



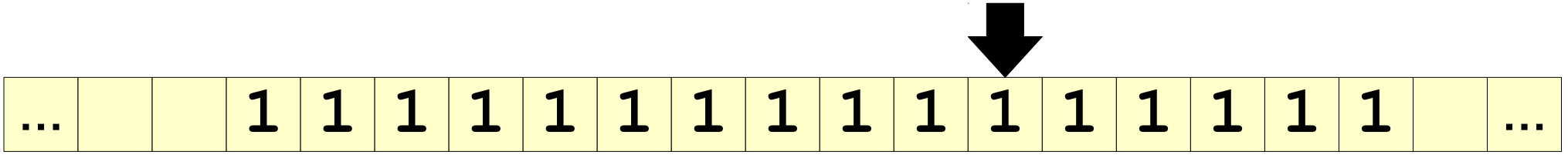
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



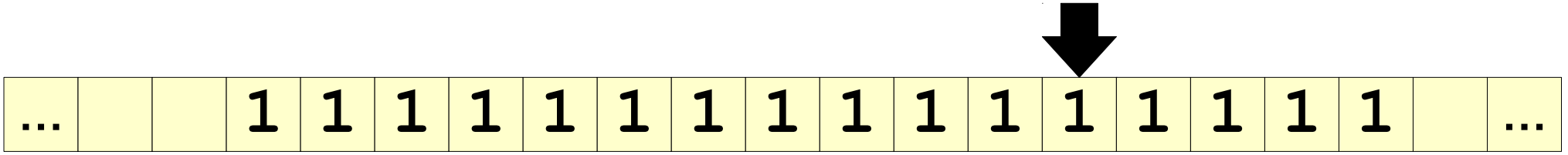
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



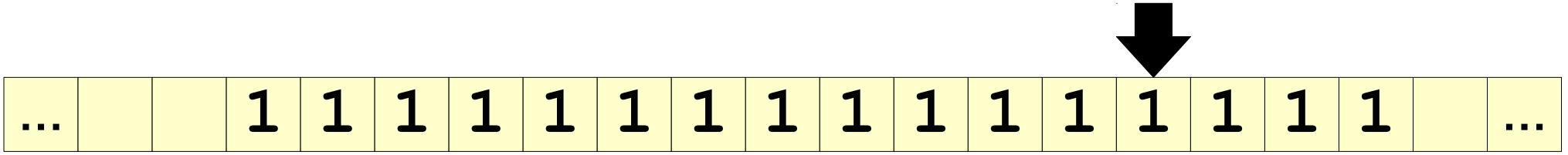
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



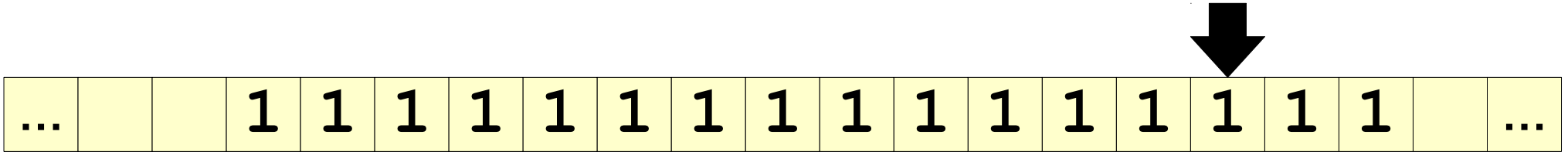
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



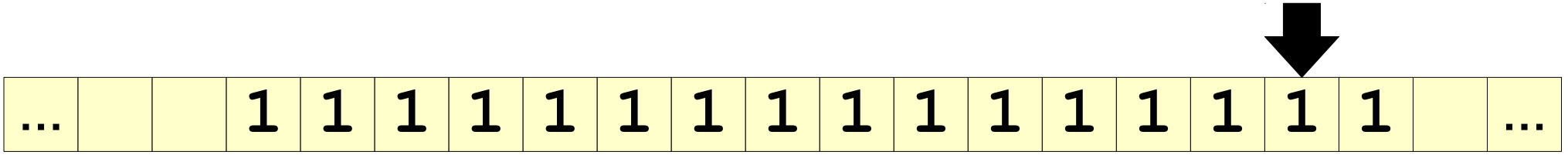
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



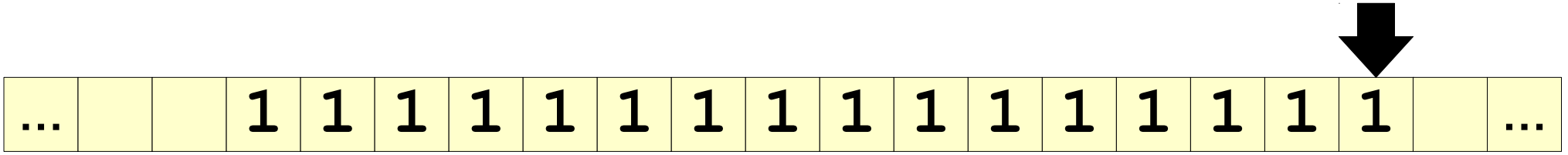
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



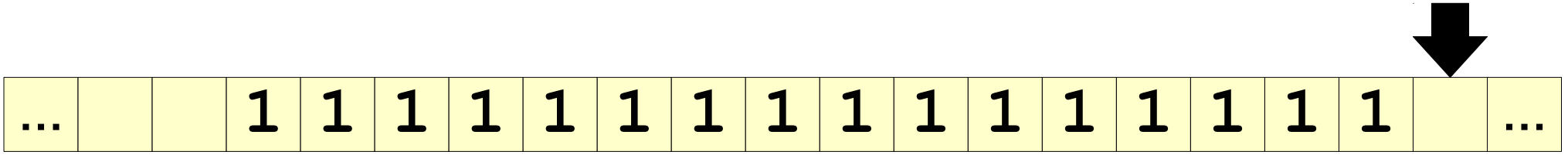
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



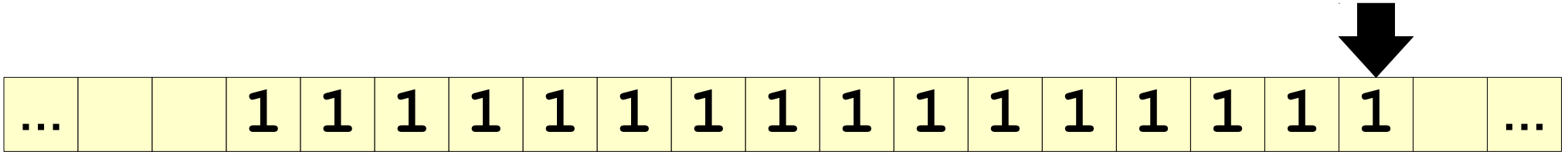
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



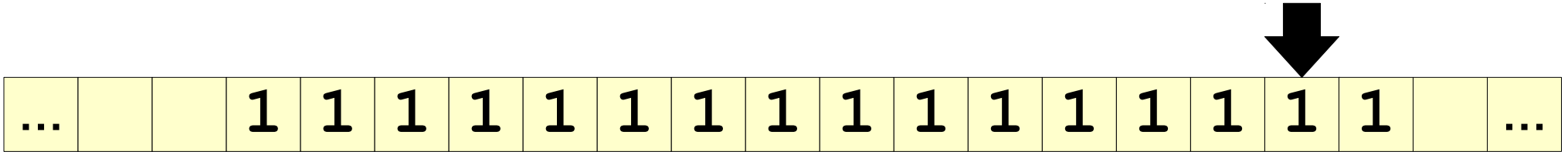
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



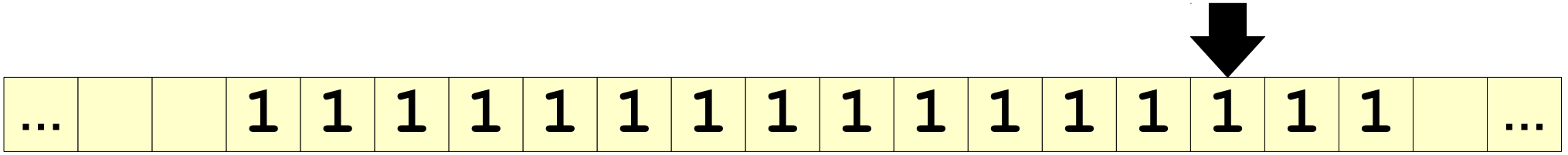
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



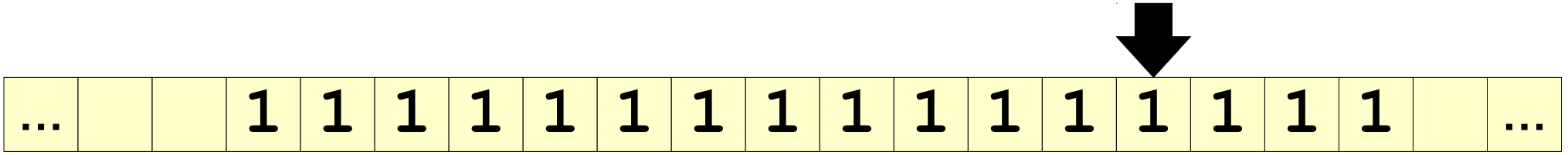
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



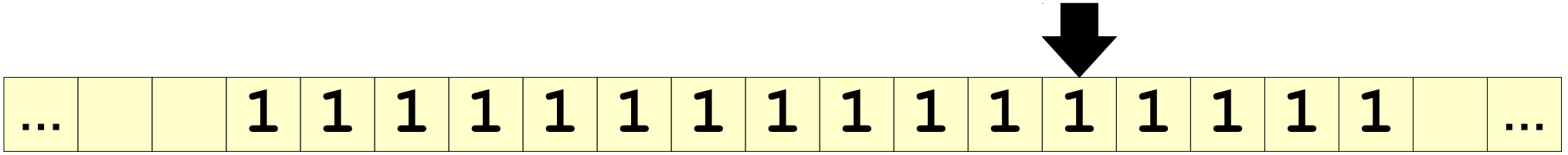
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



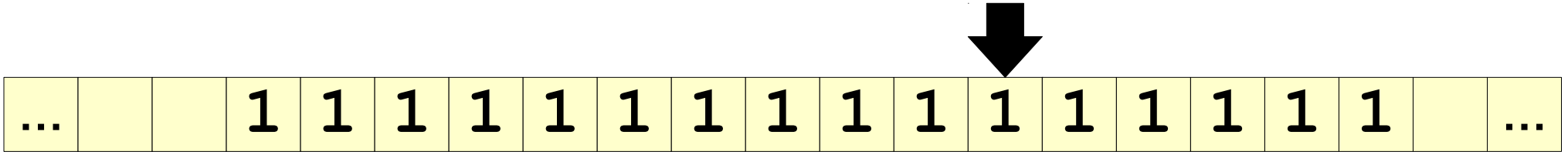
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



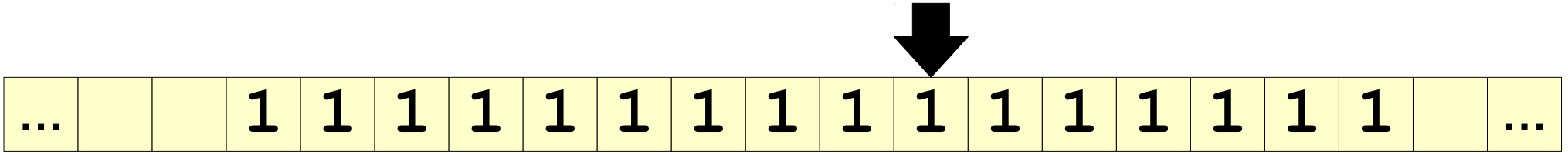
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



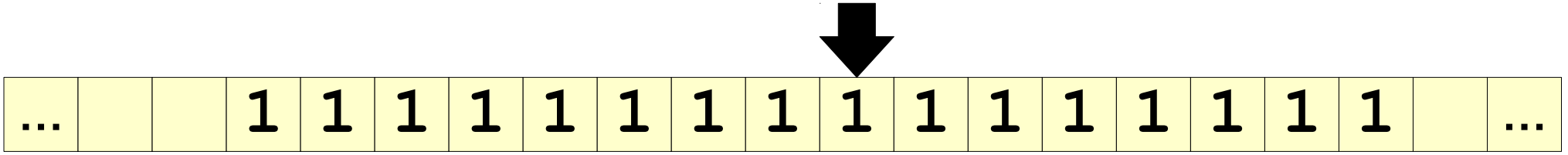
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



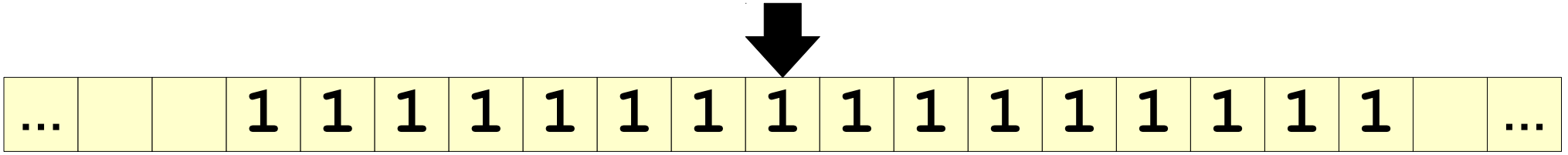
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



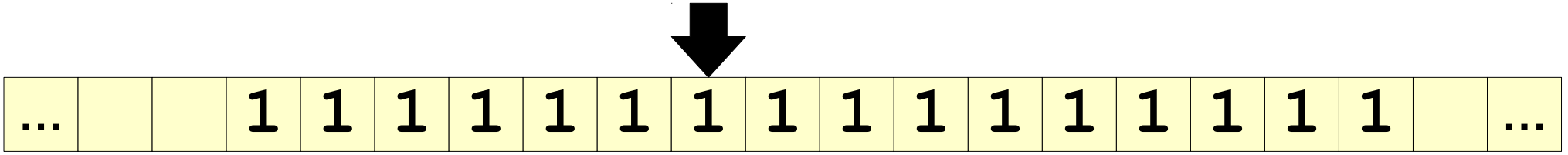
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



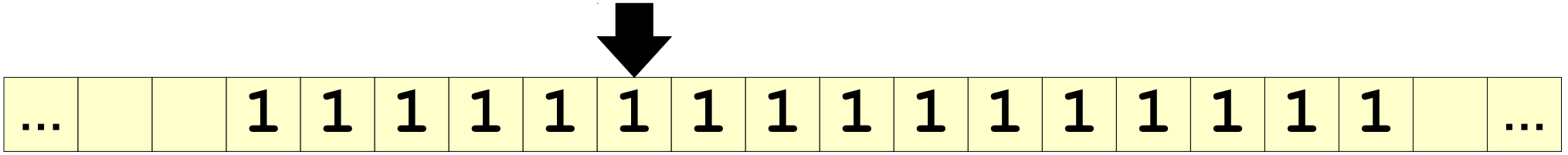
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



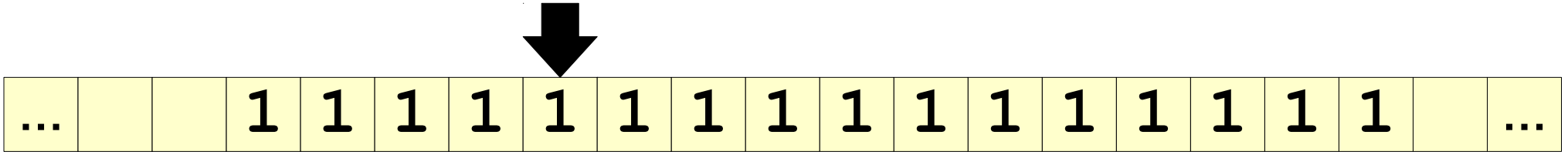
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



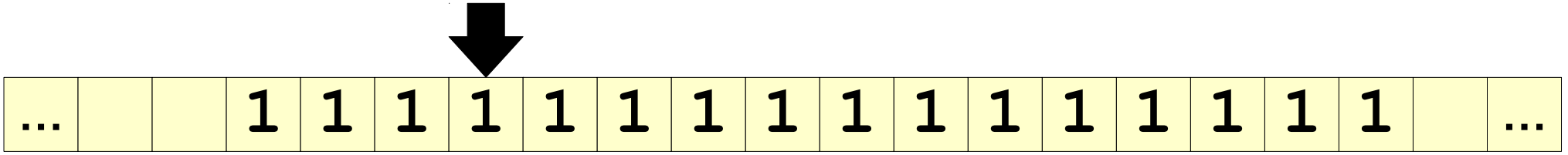
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



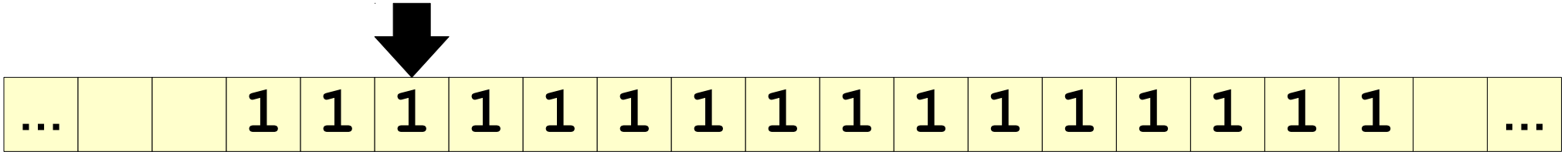
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



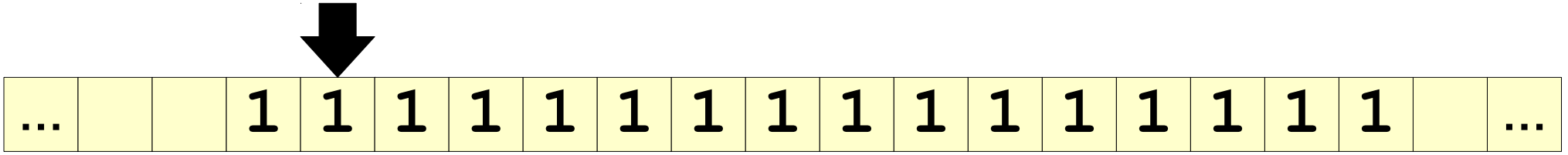
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



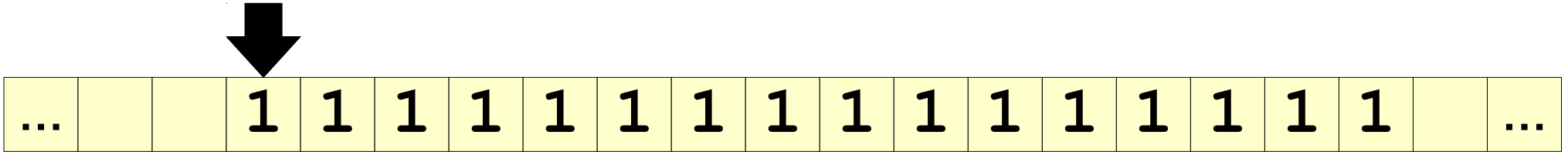
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



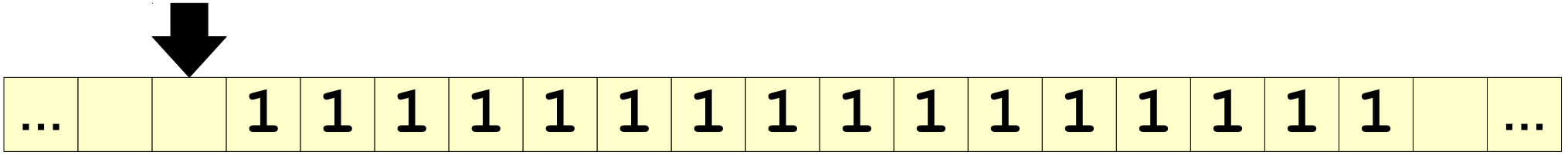
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



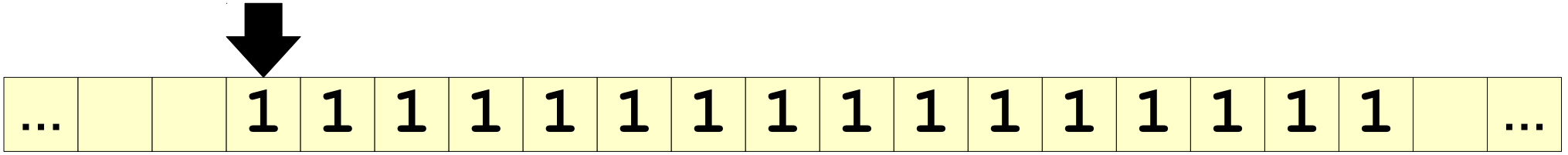
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



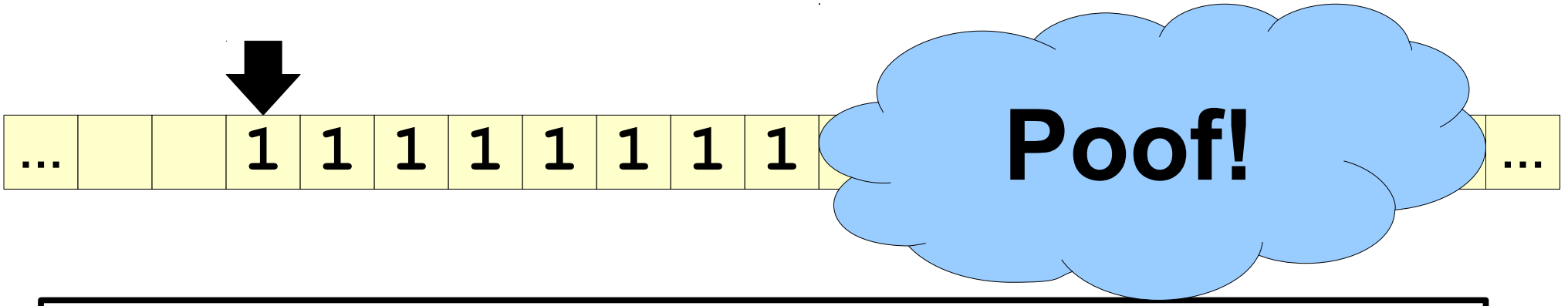
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



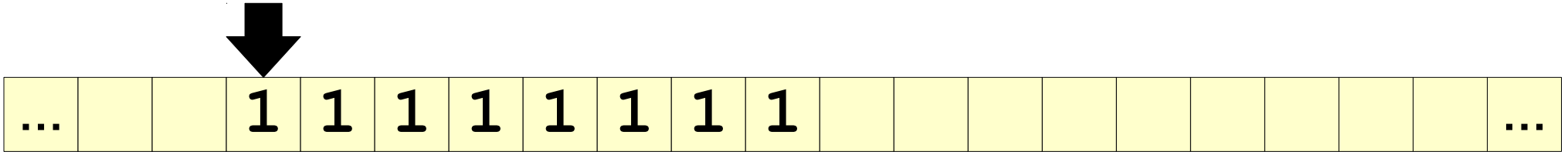
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



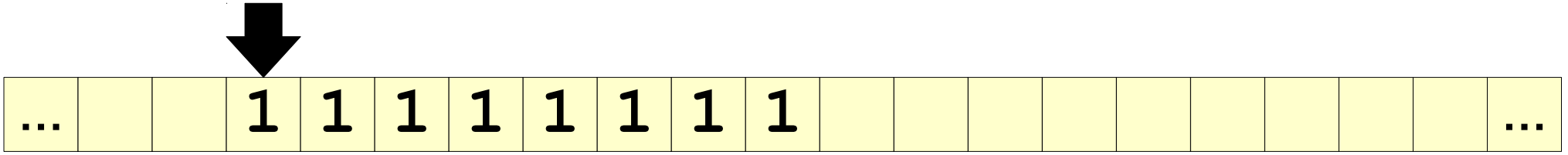
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



If the input is ε, reject.

While the input is not `1`:

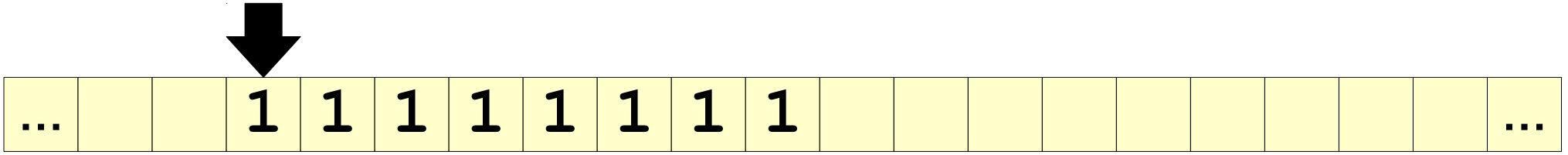
- If the input has even leng
  of the string.
- If the input has odd lengt
  string and append a `1`.

Accept.

Problem Set Question:

Build a TM that, starting with $2n$ `1`s on its tape, ends with n `1`s on its tape.

The Hailstone Turing Machine



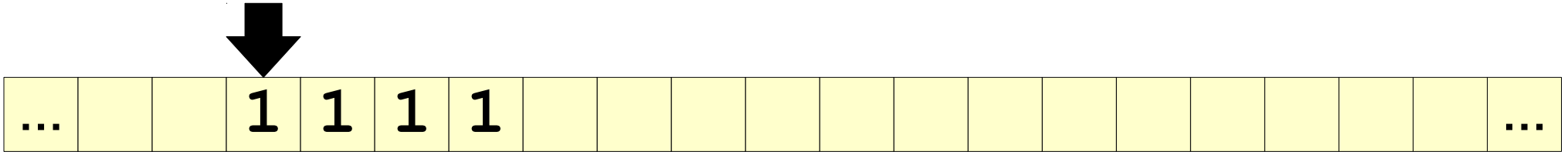
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



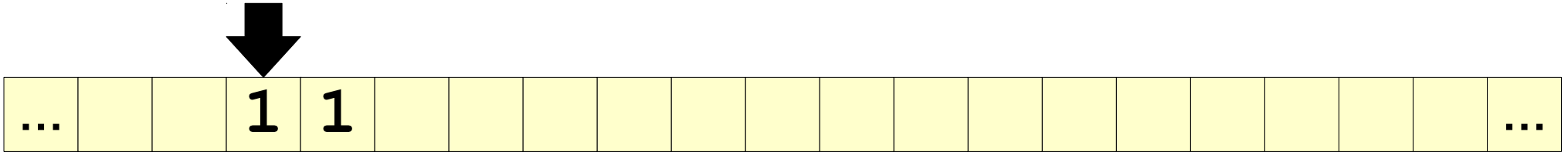
If the input is ε, reject.

While the input is not `1`:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a `1`.

Accept.

The Hailstone Turing Machine



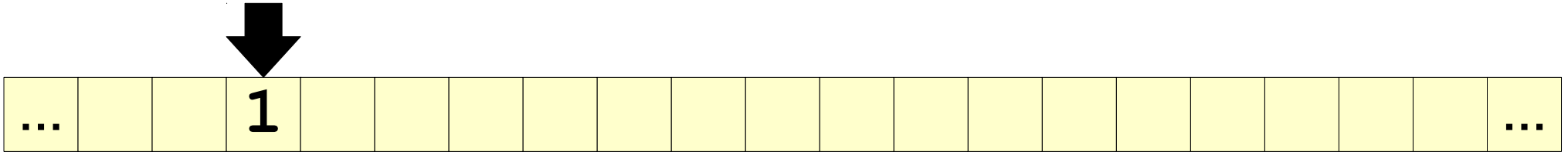
If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

The Hailstone Turing Machine



If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

Does this Turing machine accept all nonempty strings?

The Collatz Conjecture

- It is *unknown* whether this process will terminate for all natural numbers.
- In other words, no one knows whether the TM described in the previous slides will always stop running!
- The conjecture (unproven claim) that this always terminates is called the ***Collatz Conjecture***.

The Collatz Conjecture

“Mathematics may not be ready for such problems.” - Paul Erdős

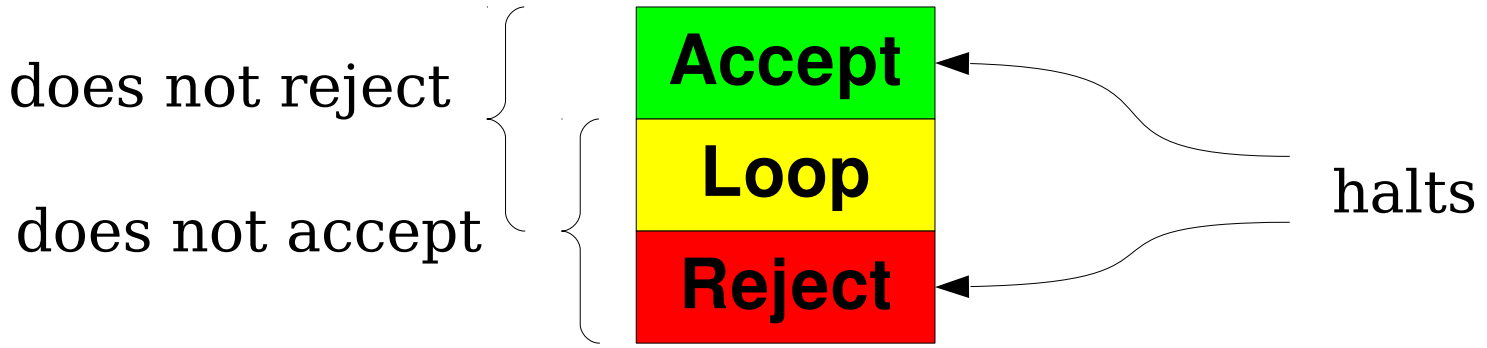
The fact that the Collatz Conjecture is unresolved is useful later on for building intuitions. Keep this in mind!

An Important Observation

- Unlike finite automata, which automatically halt after all the input is read, TMs keep running until they explicitly enter an accept or reject state.
- As a result, it’s possible for a TM to run forever without accepting or rejecting.
- This leads to several important questions:
  - How do we formally define what it means to build a TM for a language?
  - What implications does this have about problem-solving?

Very Important Terminology

- Let M be a Turing machine and let w be a string.
- M ***accepts w*** if it enters an accept state when run on w.
- M ***rejects w*** if it enters a reject state when run on w.
- M ***loops infinitely on w*** (or just ***loops on w***) if when run on w it enters neither an accept nor a reject state.
- M ***does not accept w*** if it either rejects w or loops infinitely on w.
- M ***does not reject w*** w if it either accepts w or loops on w.
- M ***halts on w*** if it accepts w or rejects w.



The Language of a TM

- The language of a Turing machine M, denoted $\mathscr{L}(M)$, is the set of all strings that M accepts:

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

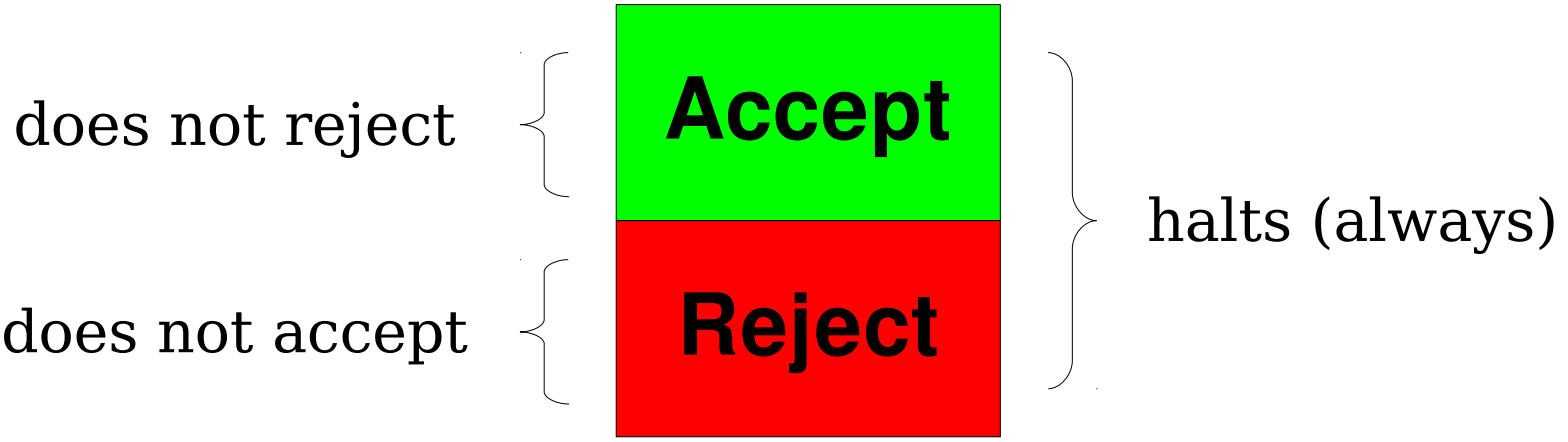
- For any $w \in \mathscr{L}(M)$, M accepts w.
- For any $w \notin \mathscr{L}(M)$, M does not accept w.
  - M might reject w, or it might loop on w.
- A language is called ***recognizable*** if it is the language of some TM.
- A TM M where $\mathscr{L}(M) = L$ is called a ***recognizer*** for L.
- Notation: the class **RE** is the set of all recognizable languages.

$$L \in \mathbf{RE} \quad \leftrightarrow \quad L \text{ is recognizable}$$

What do you think? Does that correspond to what you think it means to solve a problem?

Deciders

- Some Turing machines always halt; they never go into an infinite loop.
- If M is a TM and M halts on every possible input, then we say that M is a ***decider***.
- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.



Decidable Languages

- A language L is called ***decidable*** if there is a decider M such that $\mathscr{L}(M) = L$.
- Equivalently, a language L is decidable if there is a TM M such that
  - If $w \in L$, then M accepts w.
  - If $w \notin L$, then M rejects w.
- The class **R** is the set of all decidable languages.

$$L \in \mathbf{R} \quad \leftrightarrow \quad L \text{ is decidable}$$

- Decidable problems, in some sense, problems that can definitely be “solved” by a computer.

A Feel for **R** and **RE**

- Say you're working on a CS assignment and you ask yourself the question "does my program have a bug?"
  - An **RE** perspective: if you find a bug, you know for sure the answer is "yes", but not finding one doesn't necessarily mean the answer is "no".
  - An **R** perspective: it would be *great* if there were a magic program that could look at your code and tell you whether it's correct. *(Does something like this exist?)*

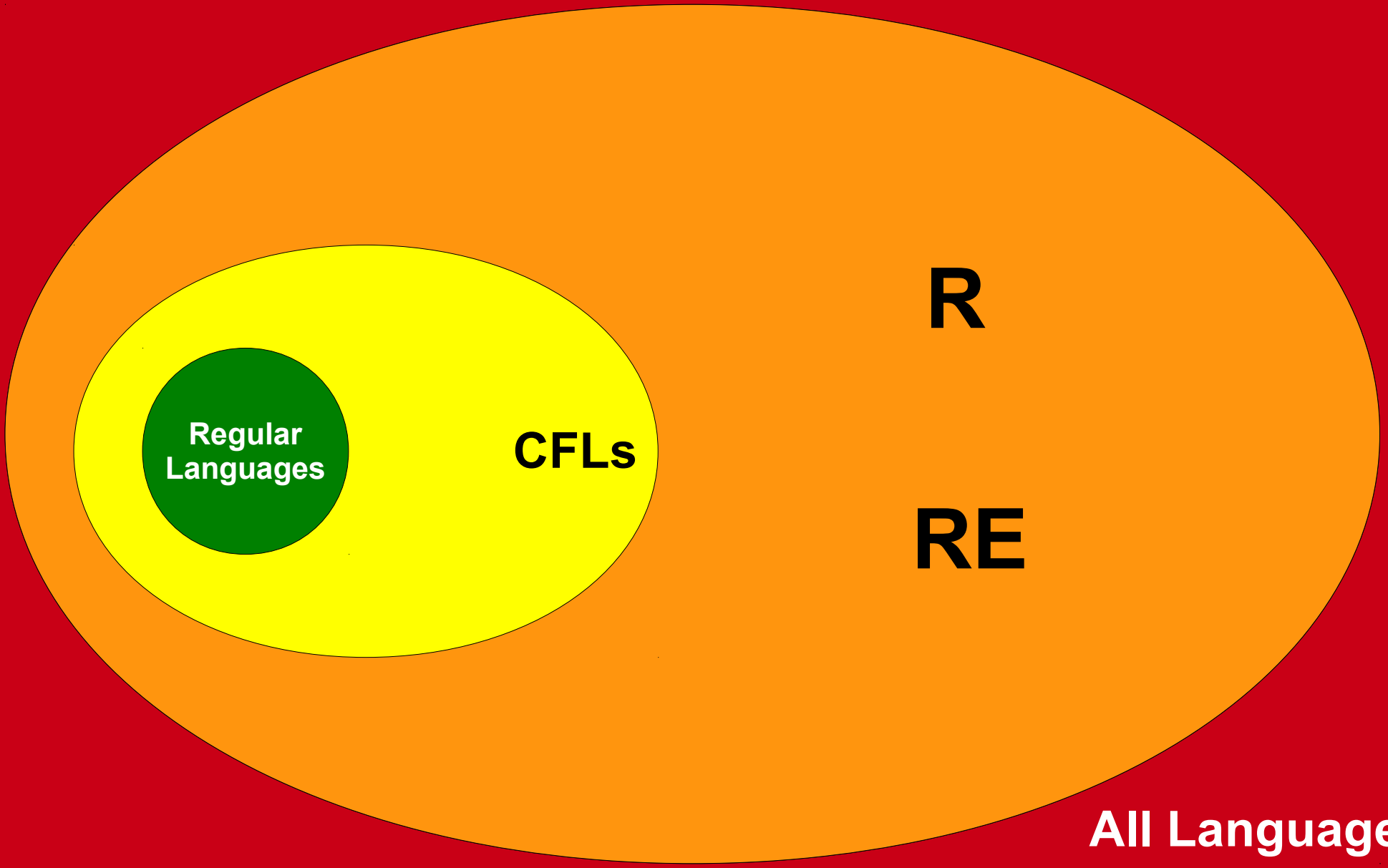
R and **RE** Languages

- Every decider is a Turing machine, but not every Turing machine is a decider.
- This means that $\mathbf{R} \subseteq \mathbf{RE}$.
- Hugely important theoretical question:

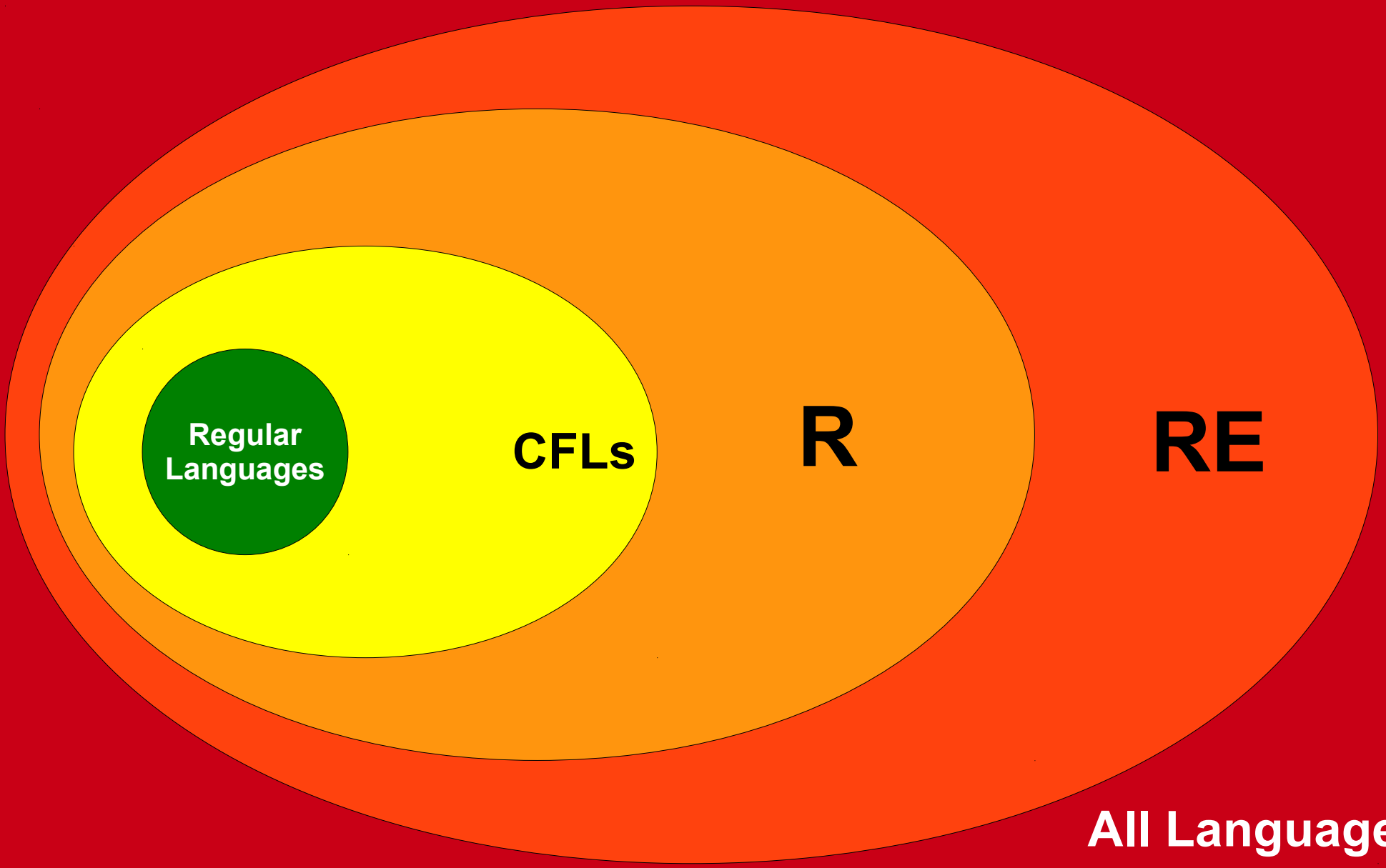
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- That is, if you can just confirm "yes" answers to a problem, can you necessarily *solve* that problem?

Which Picture is Correct?



Which Picture is Correct?



Time-Out for Announcements!

Second Midterm Graded

- We've finished grading the second midterm exam. Scores were mailed out over the break, and we'll distribute hardcopies of exams after class.
- Be sure to read over the solutions – there's a ton of goodies in there.
- Didn't pick up your exam today? No worries! They'll be sitting in a well-marked filing cabinet in the Gates building.

Problem Set Eight

- Problem Set Eight is due this Friday at 2:30PM.
  - Again, you are *encouraged* to ask questions. Stop by our office hours, or visit Piazza!
- Problem Set Seven is being graded right now. Solutions are up on the course website.

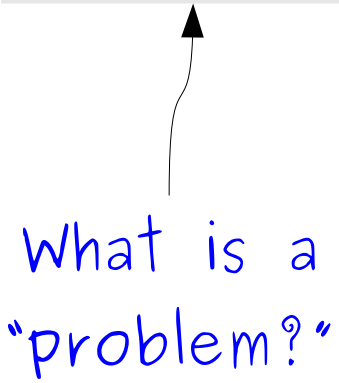
Your Questions

~~Your Questions~~

Next time!

Back to CS103!

What problems can we solve with a computer?



Decision Problems

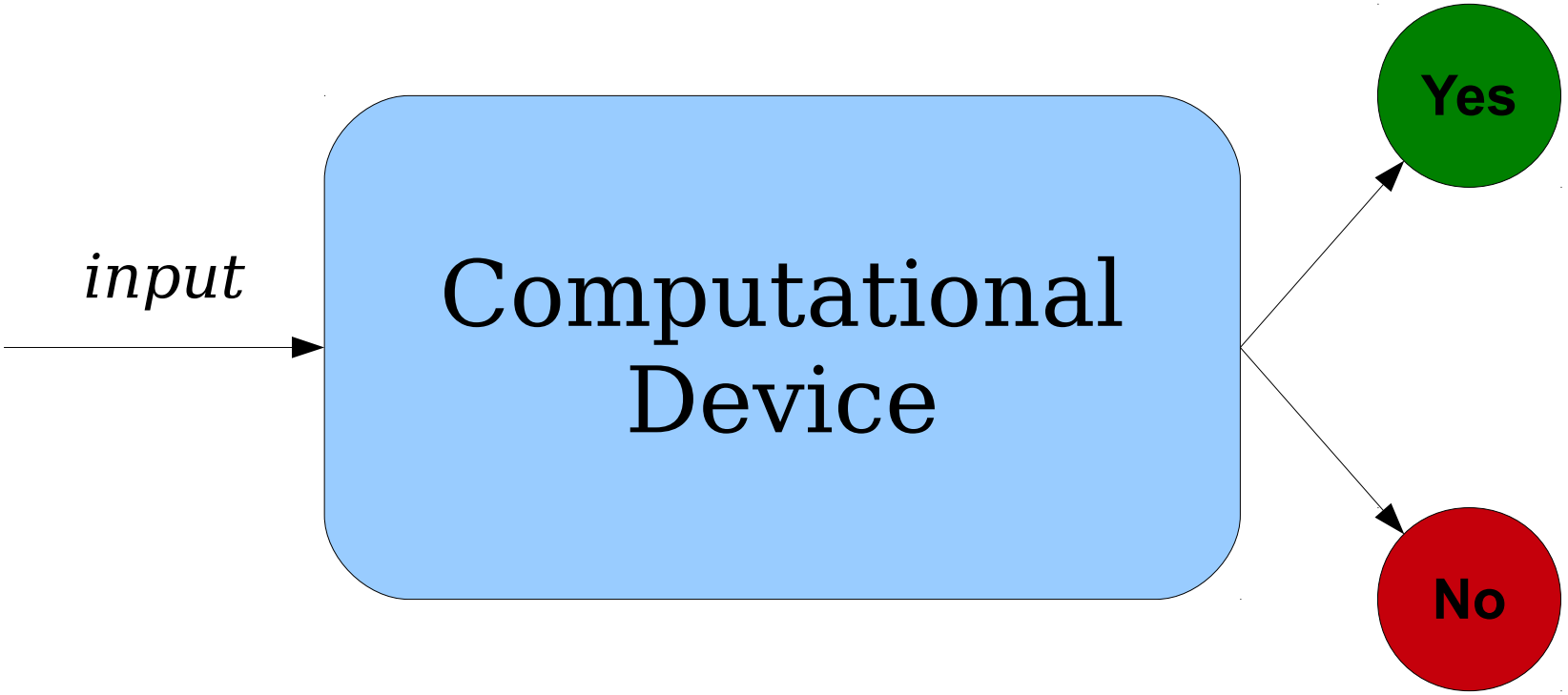
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Bin Packing

  You're given a list of patients who need to be seen and how much time each one needs to be seen for. You're given a list of doctors and how much free time they have. Is there a way to schedule the patients so that they can all be seen?

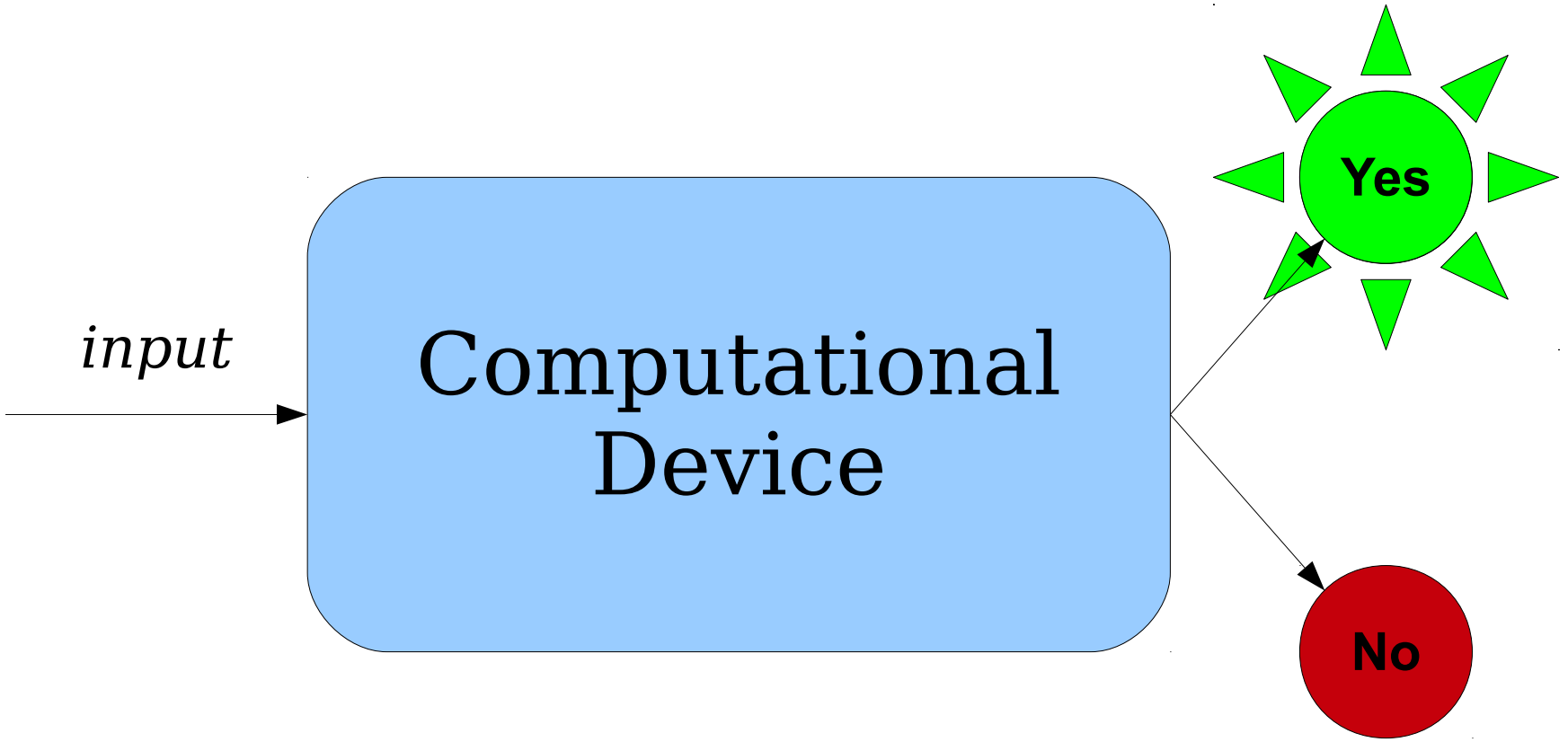
- Example: Dominating Set Problem

  You're given a transportation grid and a number k. Is there a way to place emergency supplies in at most k cities so that every city either has emergency supplies or is adjacent to a city that has emergency supplies?

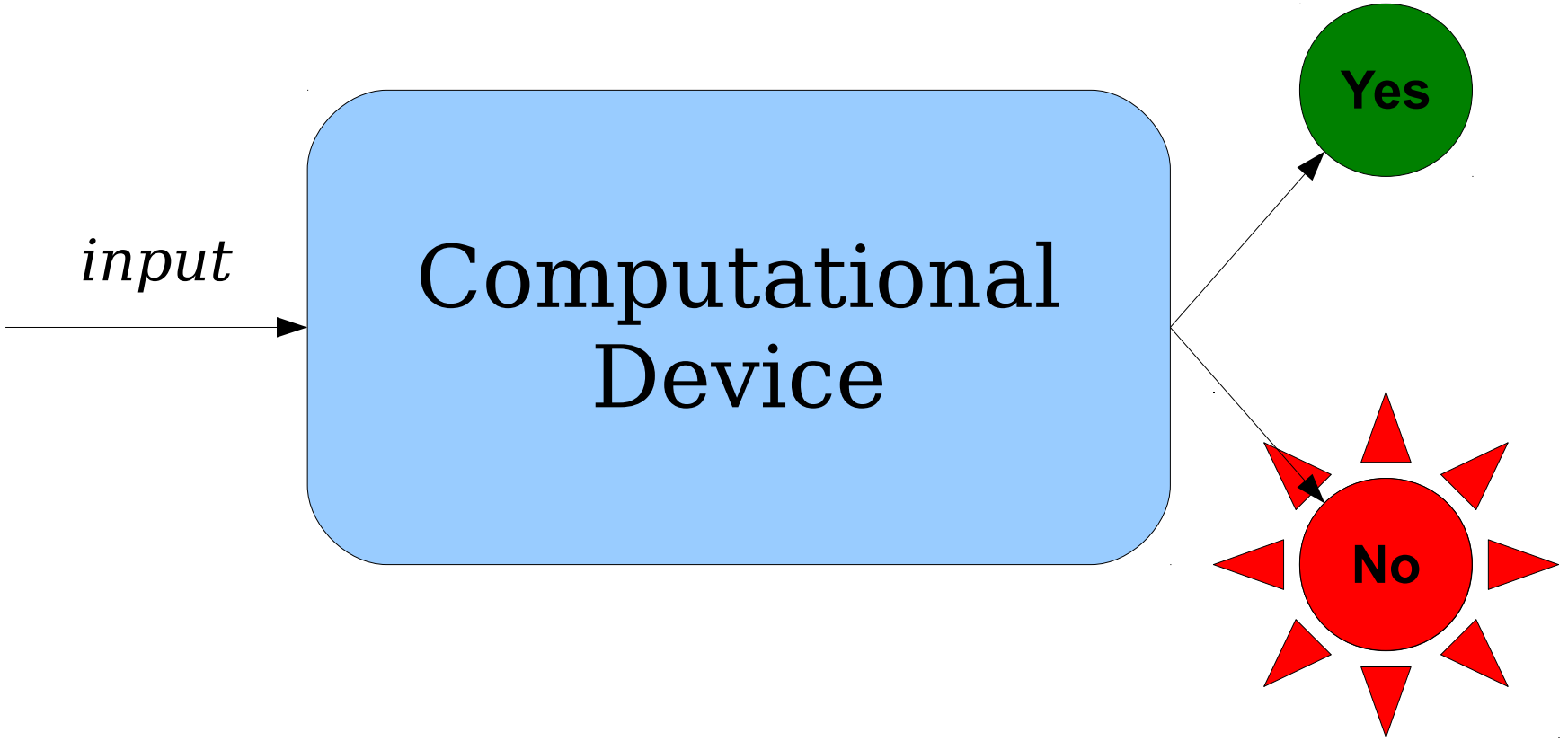
A Model for Solving Problems



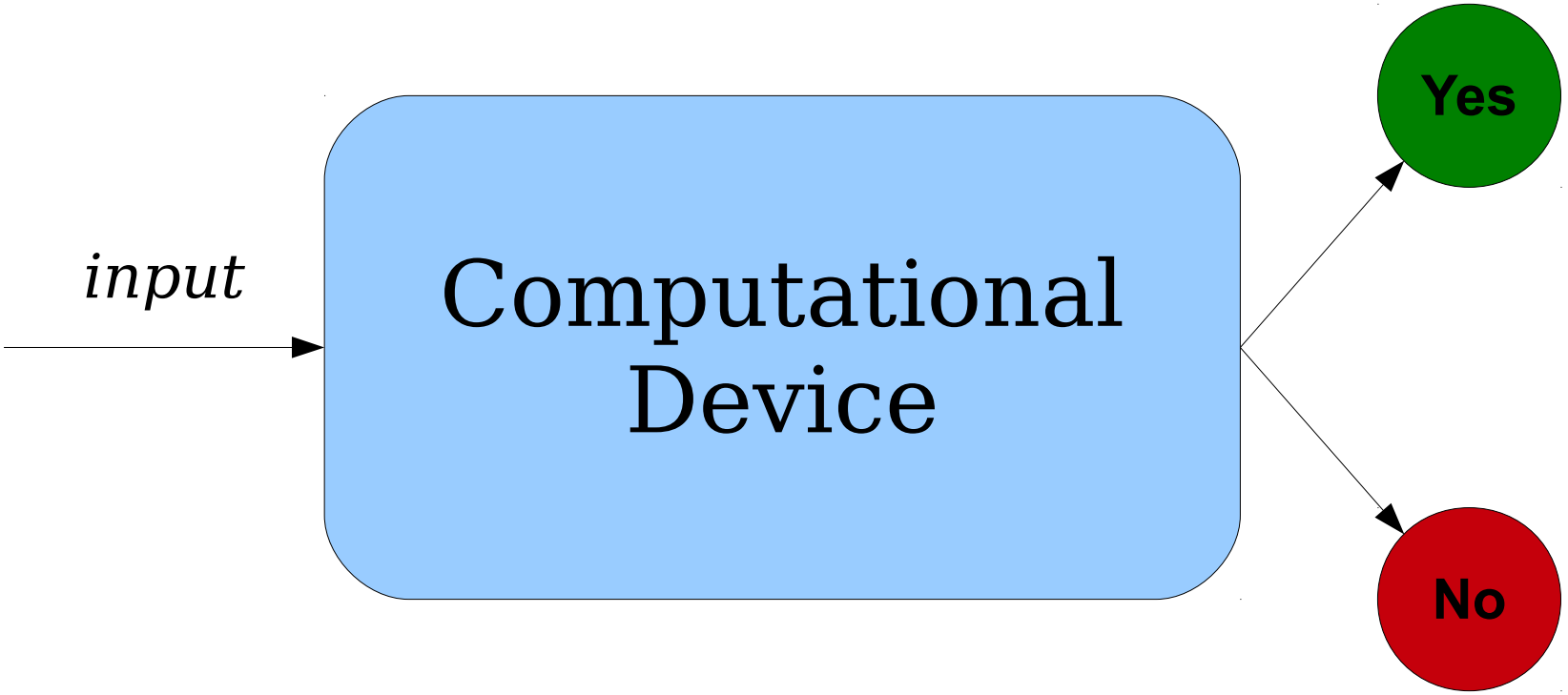
A Model for Solving Problems



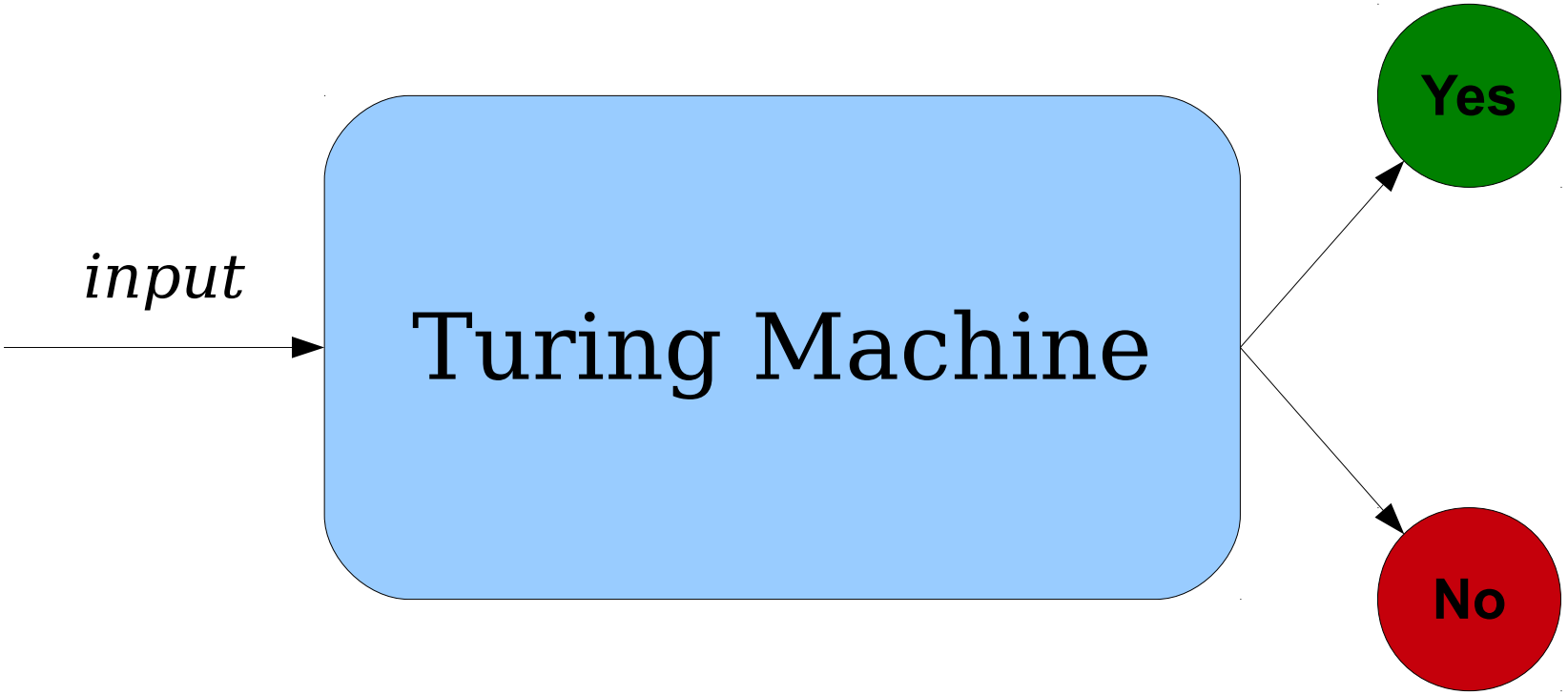
A Model for Solving Problems



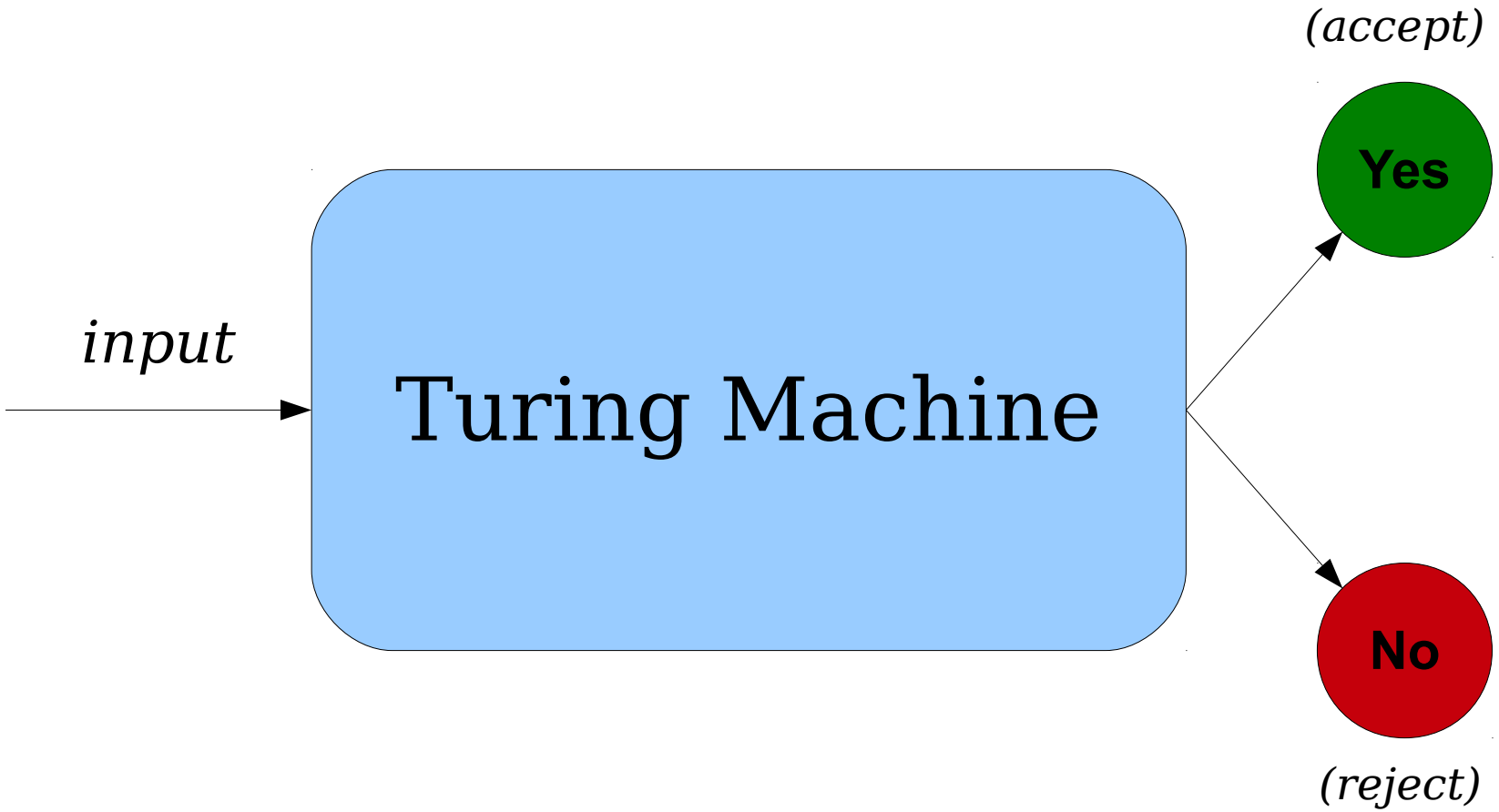
A Model for Solving Problems



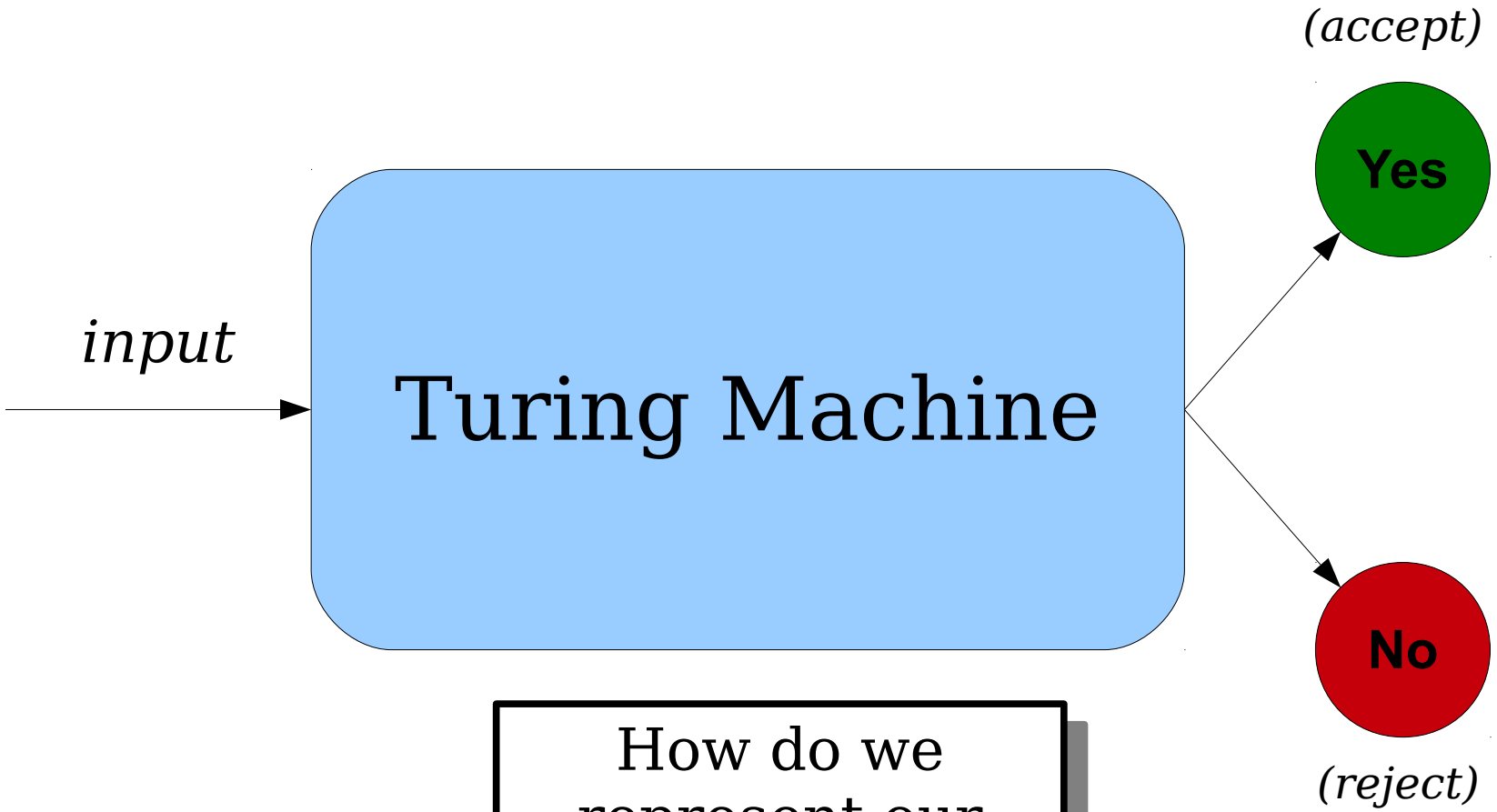
A Model for Solving Problems



A Model for Solving Problems



A Model for Solving Problems



Humbling Thought:

Everything on your computer is a string over {0, 1}.

Strings and Objects

- Think about how my computer encodes the image on the right.
- Internally, it's just a series of zeros and ones sitting on my hard drive.

Strings and Objects

- A different sequence of 0s and 1s gives rise to the image on the right.
- Every image can be encoded as a sequence of 0s and 1s, though not all sequences of 0s and 1s correspond to images.

Object Encodings

- If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation ***⟨Obj⟩*** to refer to some way of encoding that object as a string.
- Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 1101110010111011110001001 1…110

Object Encodings

- If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation ***⟨Obj⟩*** to refer to some way of encoding that object as a string.
- Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 00110101000101000101000100…001

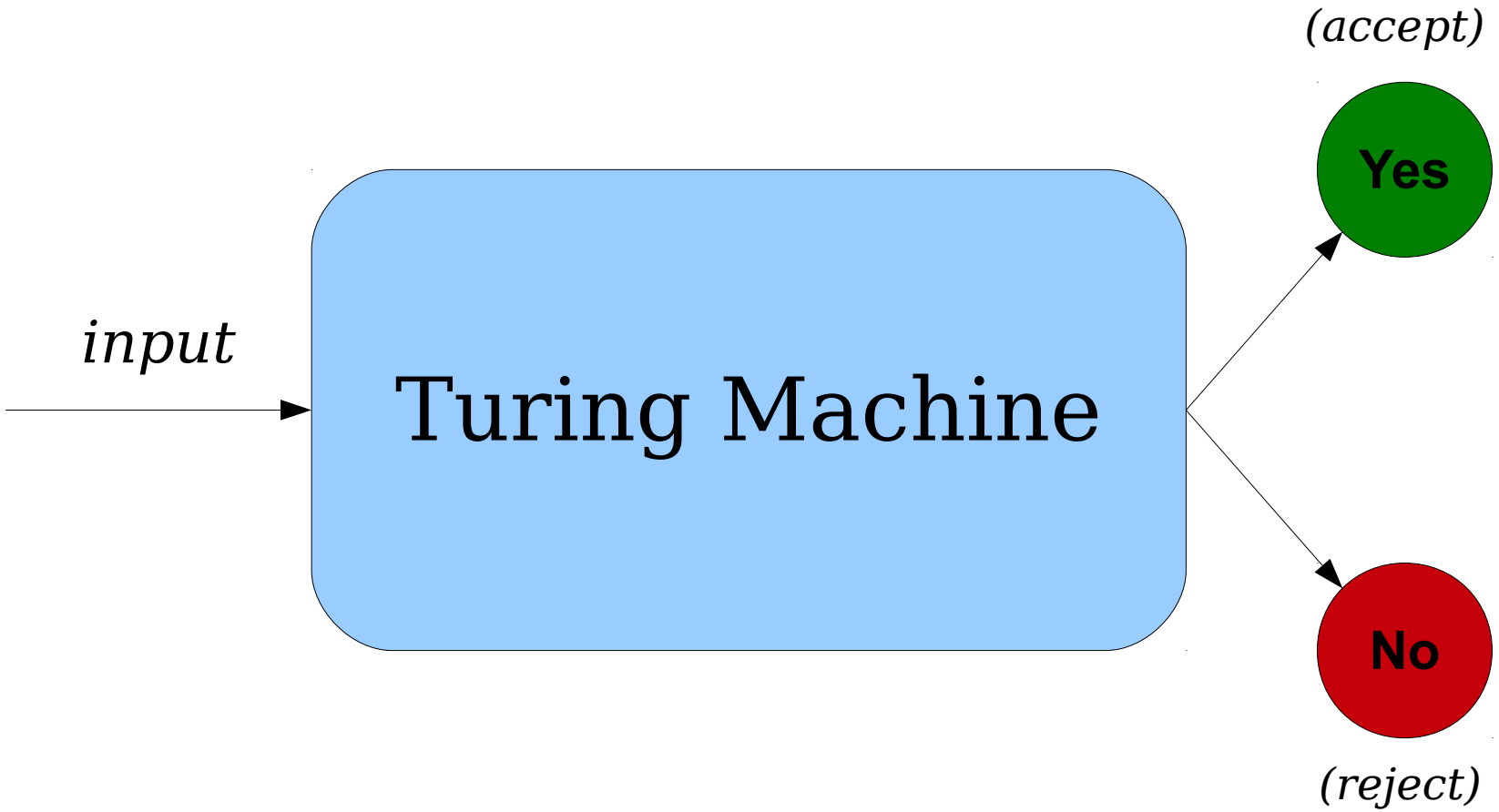
Object Encodings

- For the purposes of what we're going to be doing, we aren't going to worry about exactly *how* objects are encoded.
- For example, we can say ⟨137⟩ to mean "some encoding of 137" without worrying about how it's encoded.
  - Analogy: do you need to know how the `int` type is represented in C++ to do basic C++ programming? That's more of a CS107 question.
- We'll assume, whenever we're dealing with encodings, that some Smart, Attractive, Witty person has figured out an encoding system for us and that we're using that encoding system.

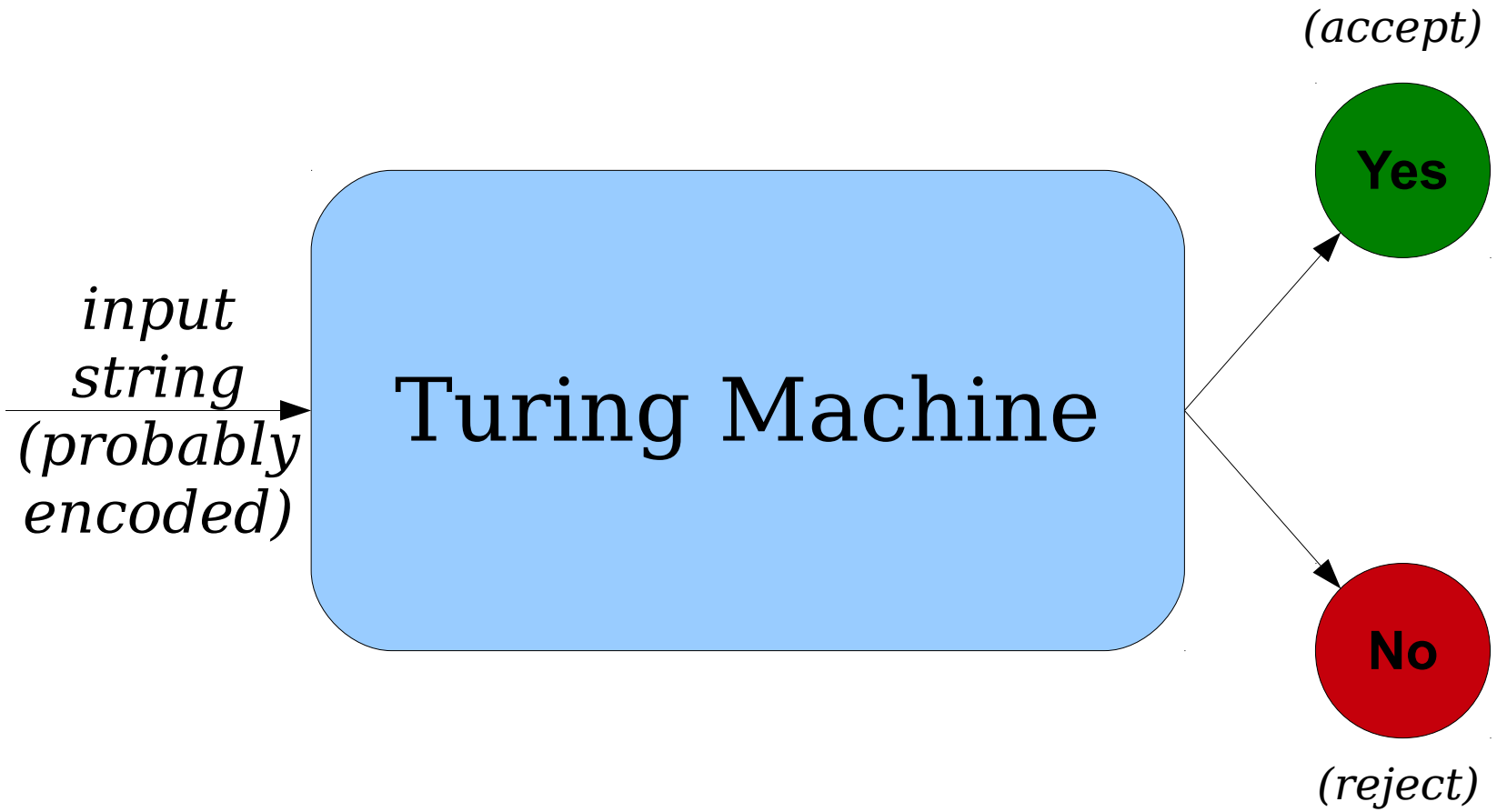
Encoding Groups of Objects

- Given a group of objects Obj_1, Obj_2, …, Obj_n, we can create a single string encoding all these objects.
  - Think of it like a .zip file, but without the compression.
- We'll denote the encoding of all of these objects as a single string by $\langle Obj_1, \ldots, Obj_n \rangle$.
- This lets us feed multiple inputs into our computational device at the same time.

A Model for Solving Problems



A Model for Solving Problems



What problems can we solve with a computer?

Emergent Properties

Emergent Properties

- An ***emergent property*** of a system is a property that arises out of smaller pieces that doesn't seem to exist in any of the individual pieces.
- Examples:
  - Individual neurons work by firing in response to particular combinations of inputs. Somehow, this leads to consciousness, love, and ennui.
  - Individual atoms obey the laws of quantum mechanics and just interact with other atoms. Somehow, it's possible to combine them together to make iPhones and pumpkin pie.

Emergent Properties of Computation

- All computing systems equal to Turing machines exhibit several surprising emergent properties.
- If we believe the Church-Turing thesis, these emergent properties are, in a sense, “inherent” to computation. Computation can’t exist without them.
- These emergent properties are what ultimately make computation so interesting and so powerful.
- As we'll see, though, they're also computation's Achilles heel – they're how we find concrete examples of impossible problems.

Two Emergent Properties

- There are two key emergent properties of computation that we will discuss:
  - ***Universality***: There is a single computing device capable of performing any computation.
  - ***Self-Reference***: Computing devices can ask questions about their own behavior.
- As you'll see, the combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.

Universal Machines

An Observation

- When we've been discussing Turing machines, we've talked about designing specific TMs to solve specific problems.
- Does this match your real-world experiences? Do you have one computing device for each task you need to perform?

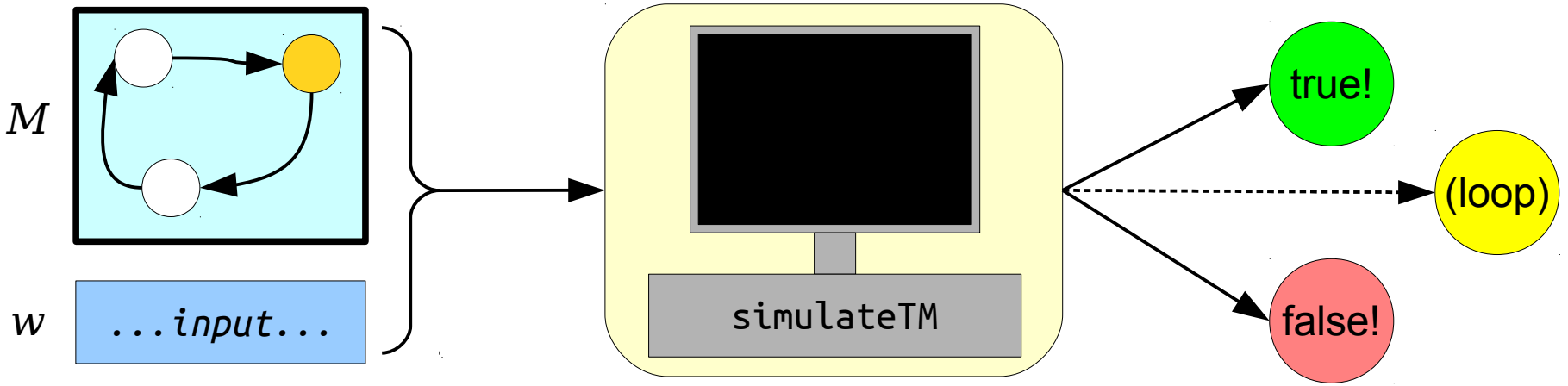
Can we make a “reprogrammable Turing machine?”

A TM Simulator

- It is possible to program a TM simulator on an unbounded-memory computer.
  - In fact, we did this! It's on the CS103 website.
- We could imagine it as a method

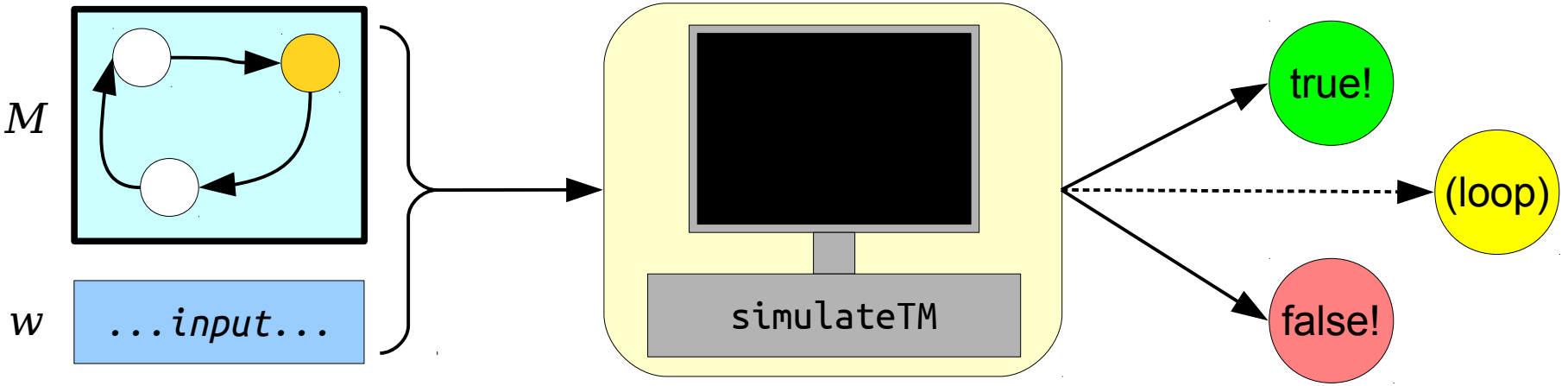
```
boolean simulateTM(TM M, string w)
```

  with the following behavior:
  - If M accepts w, then `simulateTM(M, w)` returns **`true`**.
  - If M rejects w, then `simulateTM(M, w)` returns **`false`**.
  - If M loops on w, then `simulateTM(M, w)` loops infinitely.



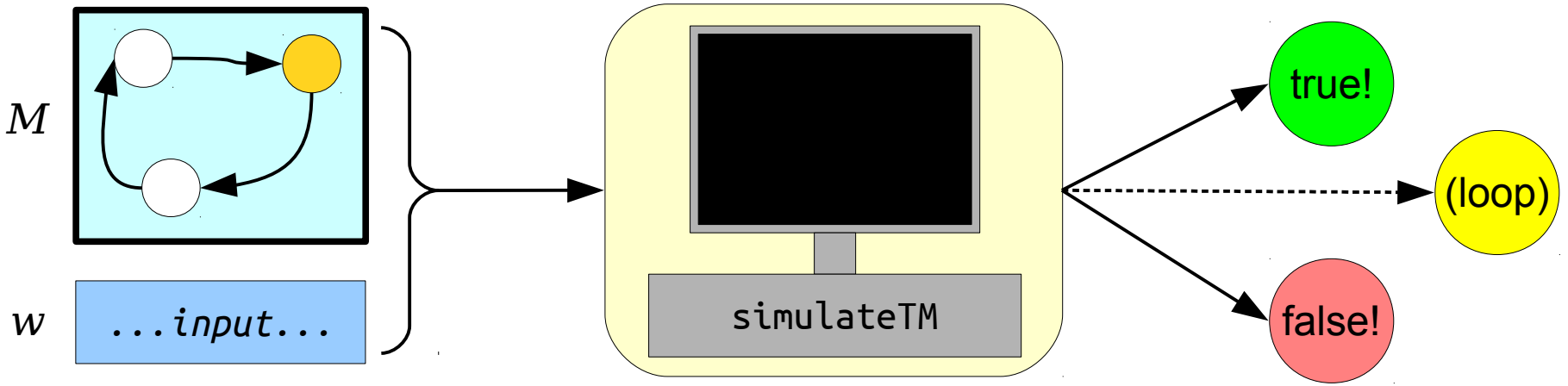
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.



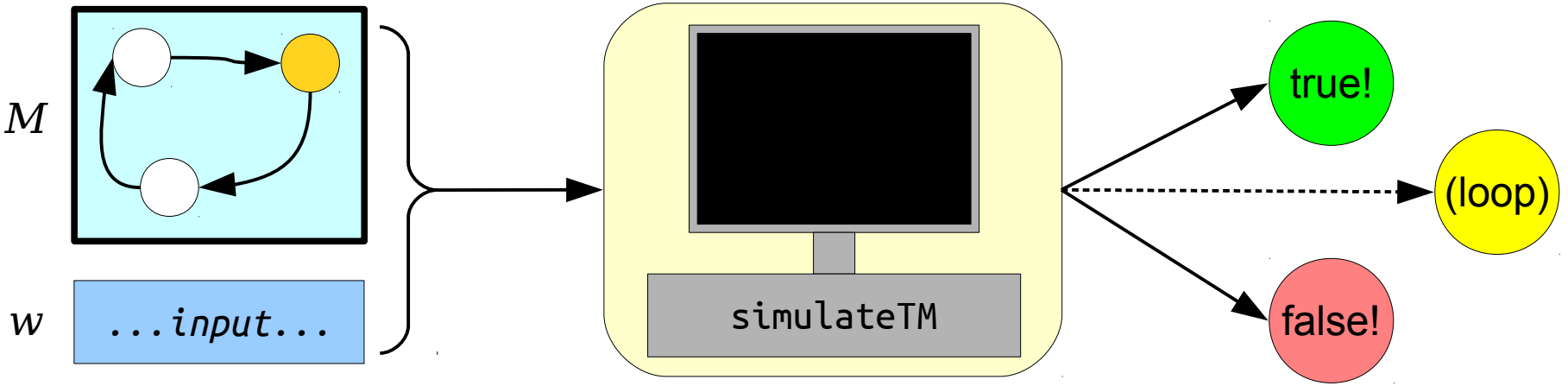
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.



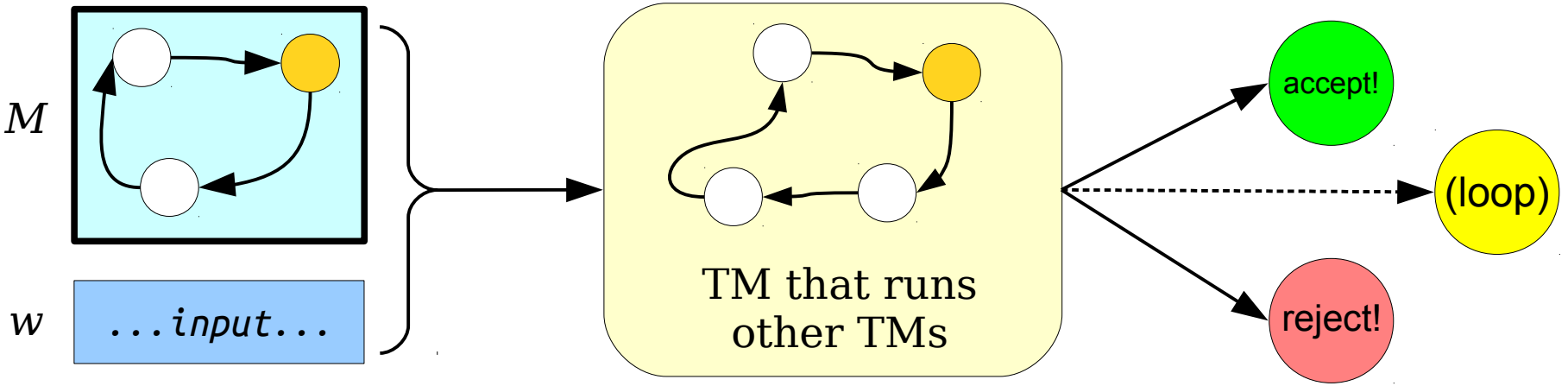
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



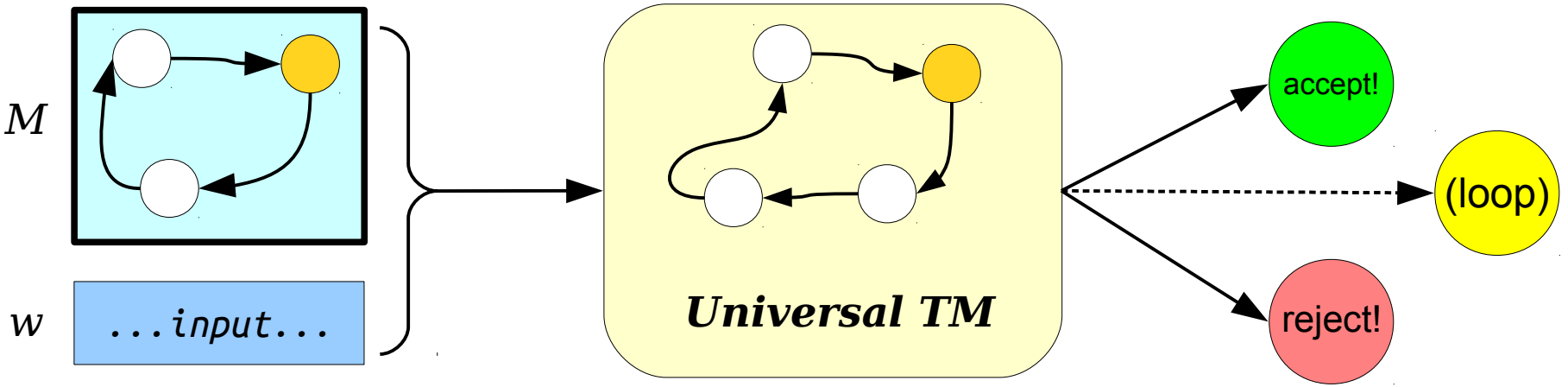
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



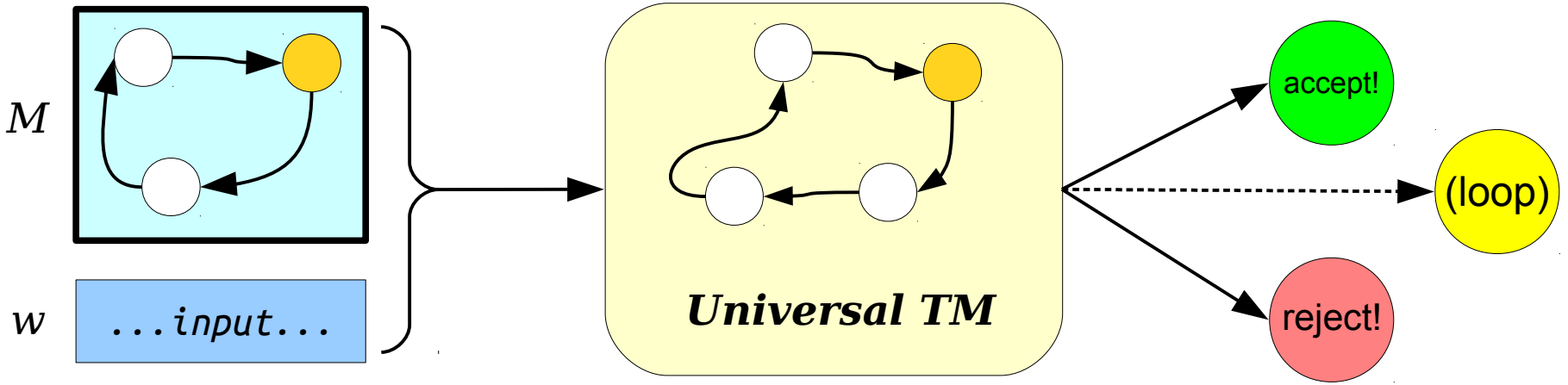
The Universal Turing Machine

- ***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 $\langle M, w \rangle$.



Since U_{TM} is a TM, it has a language.

What is the language of the universal Turing machine?

The Language of U_{TM}

- Recall that the language of a TM is the set of all strings that TM accepts.
- U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will
  - ... accept $\langle M, w \rangle$ if M accepts w,
  - ... reject $\langle M, w \rangle$ if M rejects w, and
  - ... loop on $\langle M, w \rangle$ if M loops on w.

The Language of U_{TM}

- Recall that the language of a TM is the set of all strings that TM accepts.
- U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will

  ... accept $\langle M, w \rangle$ if M accepts w,

  ~~... reject $\langle M, w \rangle$ if M rejects w, and~~

  ~~... loop on $\langle M, w \rangle$ if M loops on w.~~

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

The Language A_{TM}

- The ***acceptance language for Turing machines***, denoted $\mathbf{A_{TM}}$, is the language of the universal Turing machine:

$$A_{TM} = \mathscr{L}(U_{TM})$$

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

- Useful fact:

$$\langle M, w \rangle \in A_{TM} \quad \leftrightarrow \quad M \text{ accepts } w.$$

- Because $A_{TM} = \mathscr{L}(U_{TM})$, we know that $A_{TM} \in \mathbf{RE}$.

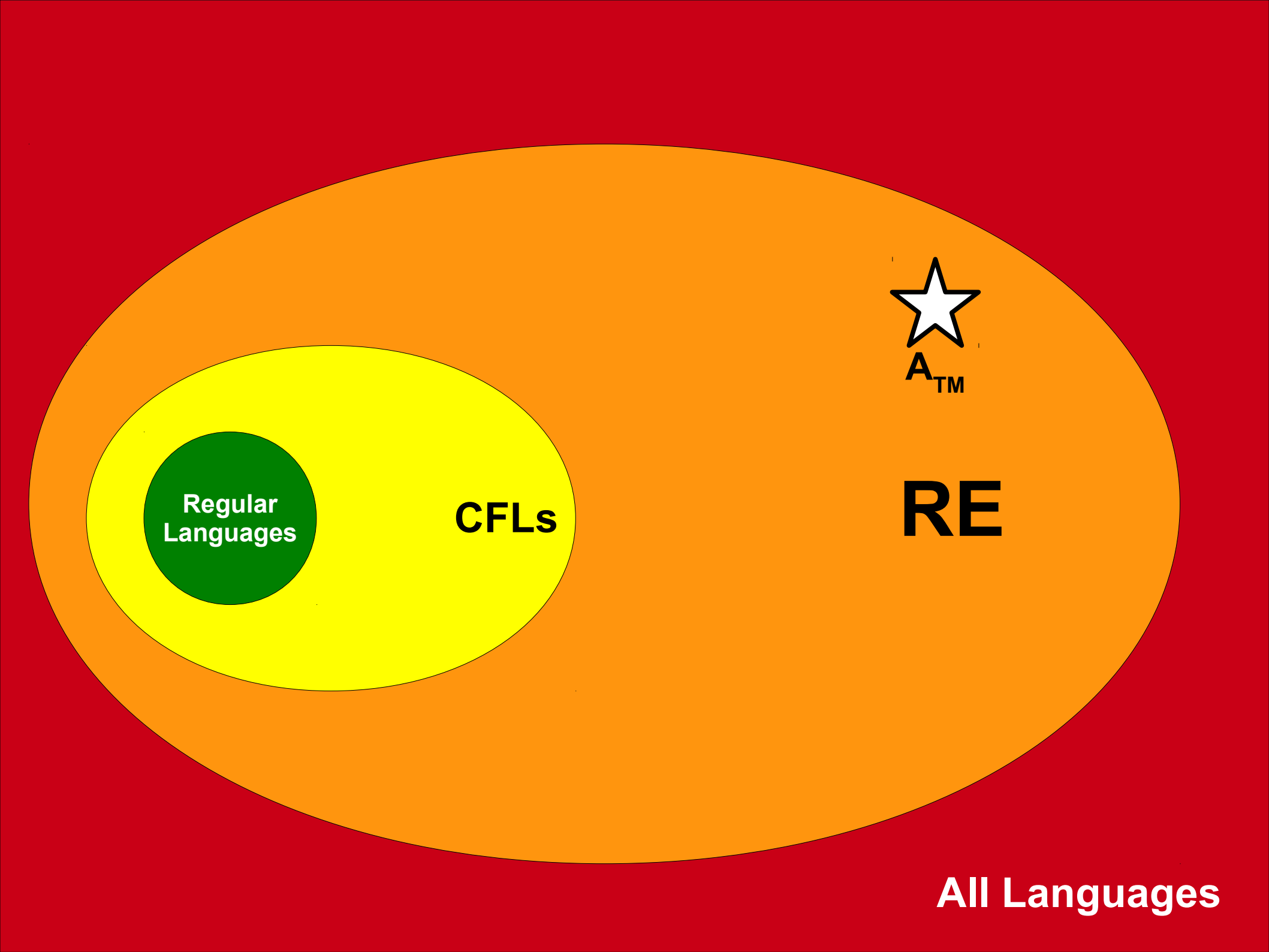
Great Question to Ponder

- Simplify this expression:

$$\langle U_{TM}, \langle U_{TM}, \langle U_{TM}, \langle U_{TM}, \langle M, w \rangle \rangle \rangle \rangle \rangle \in A_{TM}.$$

- If you can do this, you probably understand how things fit together.
- If you're having trouble, no worries! It might be easier to start with this expression:

$$\langle U_{TM}, \langle M, w \rangle \rangle \in A_{TM}.$$

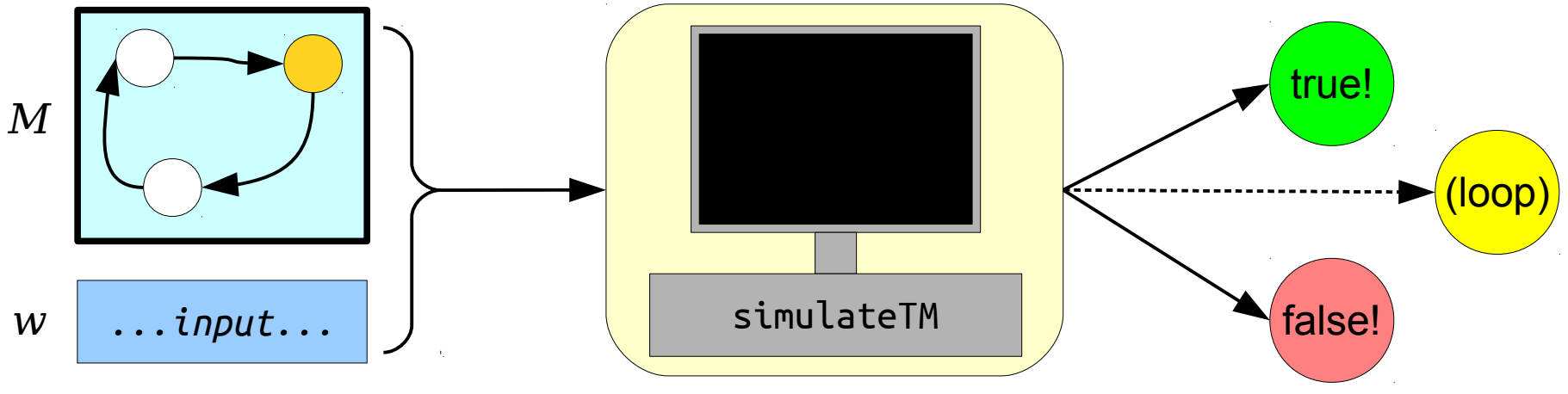

Regular Languages
CFLs
A_{TM}
RE
All Languages

Uh… so what?

Reason 1: ***It has practical consequences.***

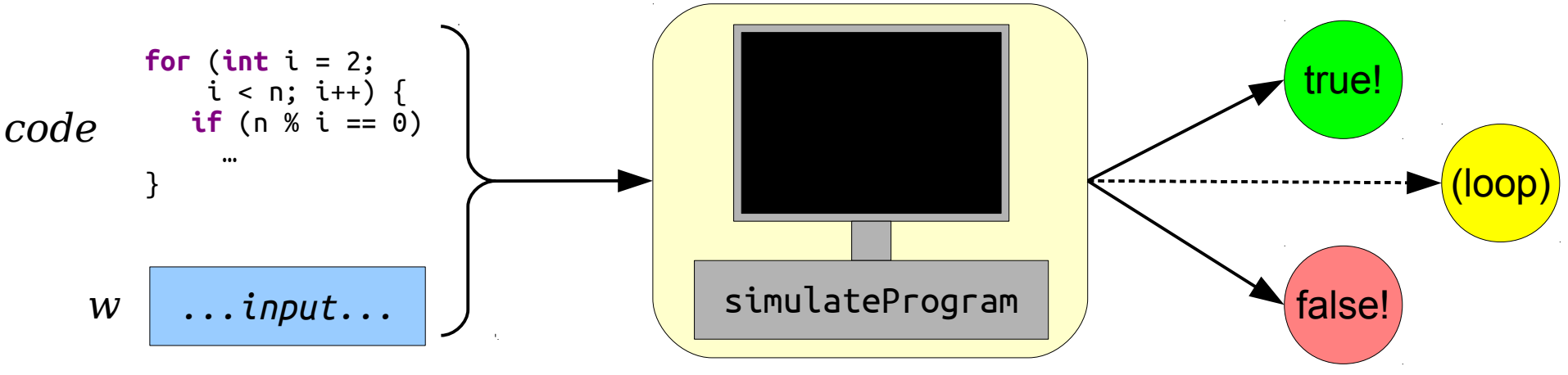
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the *computer* with a TM. (This gave us the universal TM.)
- What happens if we replace the *TM* with a computer program?



Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the *computer* with a TM. (This gave us the universal TM.)
- What happens if we replace the *TM* with a computer program?



Programs Simulating Programs

- The fact that there's a universal TM, combined with the fact that computers can simulate TMs and vice-versa, means that it's possible to write a program that simulates other programs.
- These programs go by many names:
  - An ***interpreter***, like the Java Virtual Machine or most implementations of Python.
  - A ***virtual machine***, like VMWare or VirtualBox, that simulates an entire computer.

Why Does This Matter?

- The key idea behind the universal TM is that idea that TMs can be fed as inputs into other TMs.
  - Similarly, an interpreter is a program that takes other programs as inputs.
  - Similarly, an emulator is a program that takes entire computers as inputs.
- This hits at the core idea that ***computing devices can perform computations on other computing devices.***

Reason 2: ***It's philosophically interesting.***

Can Computers Think?

- On May 15, 1951, Alan Turing delivered **a radio lecture on the BBC** on the topic of whether computers can think.
- He had the following to say about whether a computer can be thought of as an electric brain...

“In fact I think they [computers] could be used in such a manner that they could be appropriately described as brains. I should also say that

‘If any machine can be appropriately described as a brain, then any digital computer can be so described.’

This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

It can be shown to follow from a characteristic property of digital computers, which I will call their ***universality***. A digital computer is a universal machine in the sense that it can be made to replace any machine of a certain very wide class. It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results. In order to arrange for our computer to imitate a given machine it is only necessary to programme the the computer to calculate what the machine in question would do under given circumstances, and in particular what answers it would print out. The computer can then be made to print out the same answers.

If now some machine can be described as a brain we have only to programme our digital computer to imitate it and it will also be a brain.”

Next Time

- ***Self-Reference***
  - Turing machines that compute on themselves!
- ***Undecidable Problems***
  - Problems truly beyond the limits of algorithmic problem-solving!
- ***Consequences of Undecidability***
  - Why does any of this matter outside of a computer science course?