

Turing Machines

Part Three

Recap from Last Time

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
Turing
Machines**

All Languages

But...but...but...can a TM do ?!


- Play music?
 - Yes! Writes encodings of notes to play on TM tape, speaker device reads the tape and makes sound
- Send chat messages over the internet?
 - Yes! Writes encodings of messages to send on the TM tape, network device launches off message

But...but...but...can a TM do
_____?!

- Make another Turing Machine?
- Yes! That is the subject of today. :-)


What problems can we solve with a computer?

What kind of
computer?



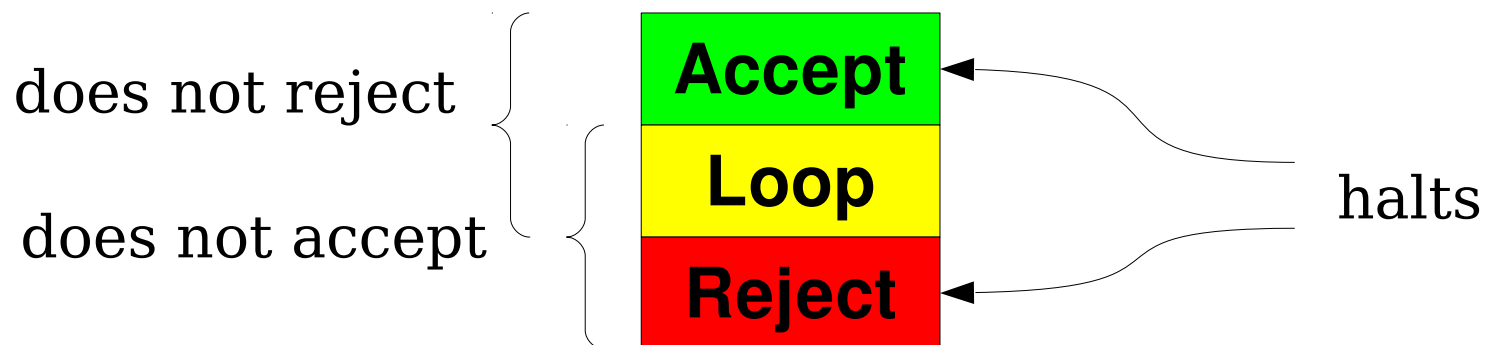
What problems can we solve with a computer?

What does it
mean to "solve"
a problem?



Very Important Terminology

- Let M be a Turing machine.
- M **accepts** a string w if it enters an accept state when run on w .
- M **rejects** a string w if it enters a reject state when run on w .
- M **loops infinitely** (or just **loops**) on a string 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** 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 $\mathcal{L}(M)$, is the set of all strings that M accepts:

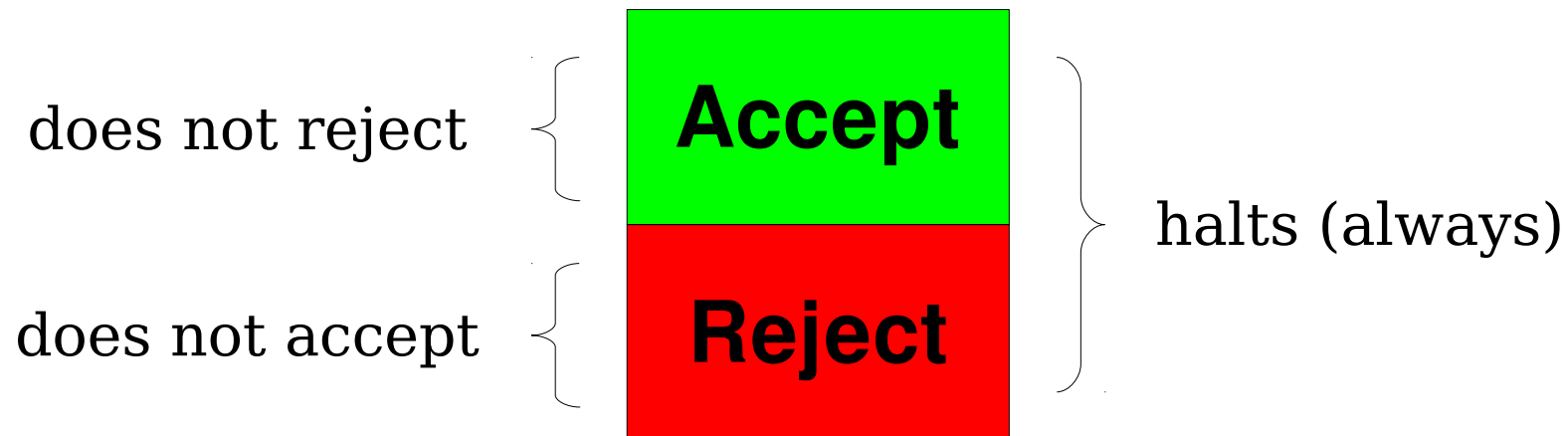
$$\mathcal{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

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

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

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 $\mathcal{L}(M) = L$.
- 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.

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

New Stuff!

Why **R** Matters

- If a language is in **R**, there is an algorithm that can decide membership in that language.
 - Run the decider and see what it says.
- If there is an algorithm that can decide membership in a language, that language is in **R**.
 - By the Church-Turing thesis, any effective model of computation is equivalent in power to a Turing machine.
 - Therefore, if there is *any* algorithm for deciding membership in the language, there is a decider for it.
 - Therefore, the language is in **R**.
- ***A language is in R if and only if there is an algorithm for deciding membership in that language.***

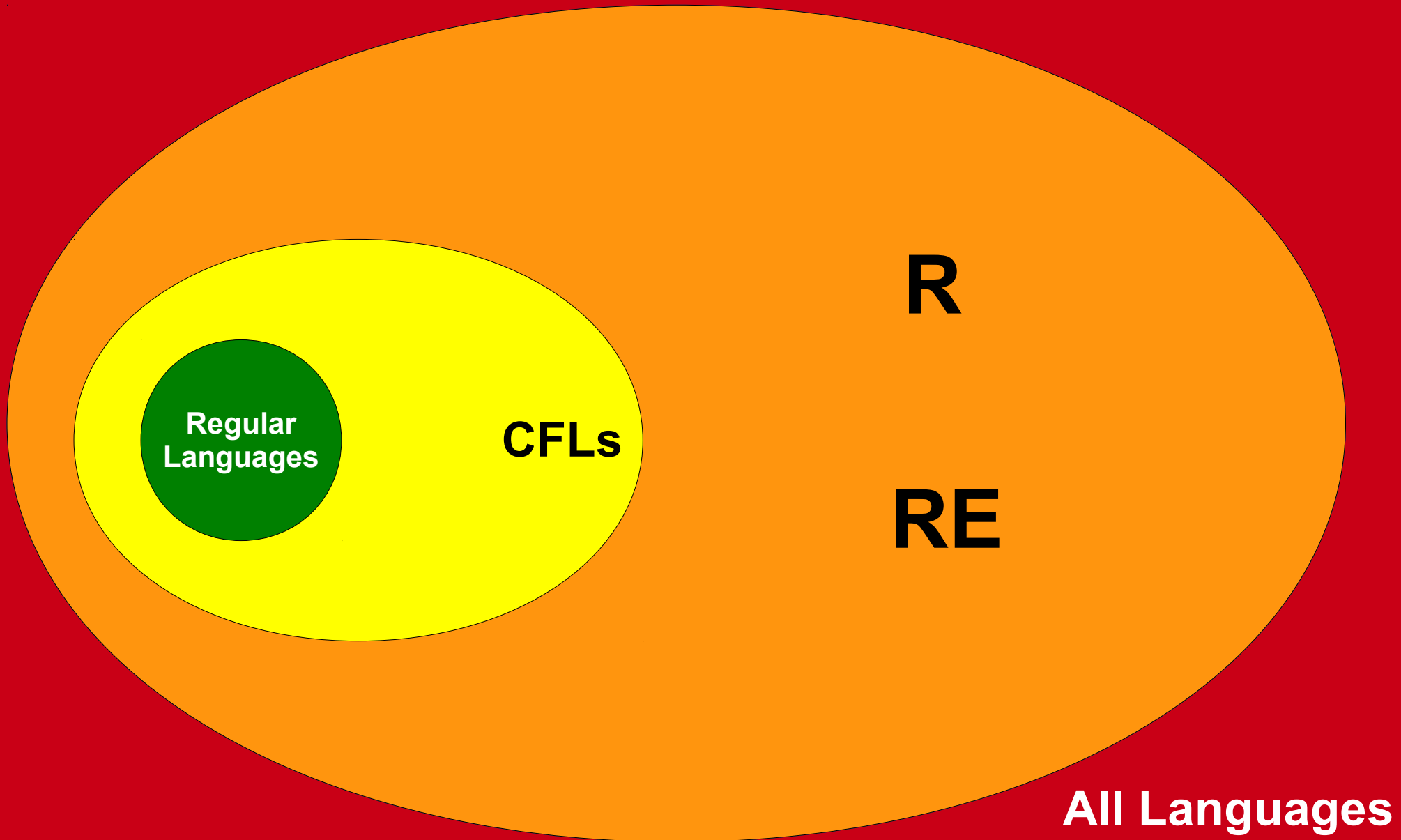
R and **RE** Languages

- 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:

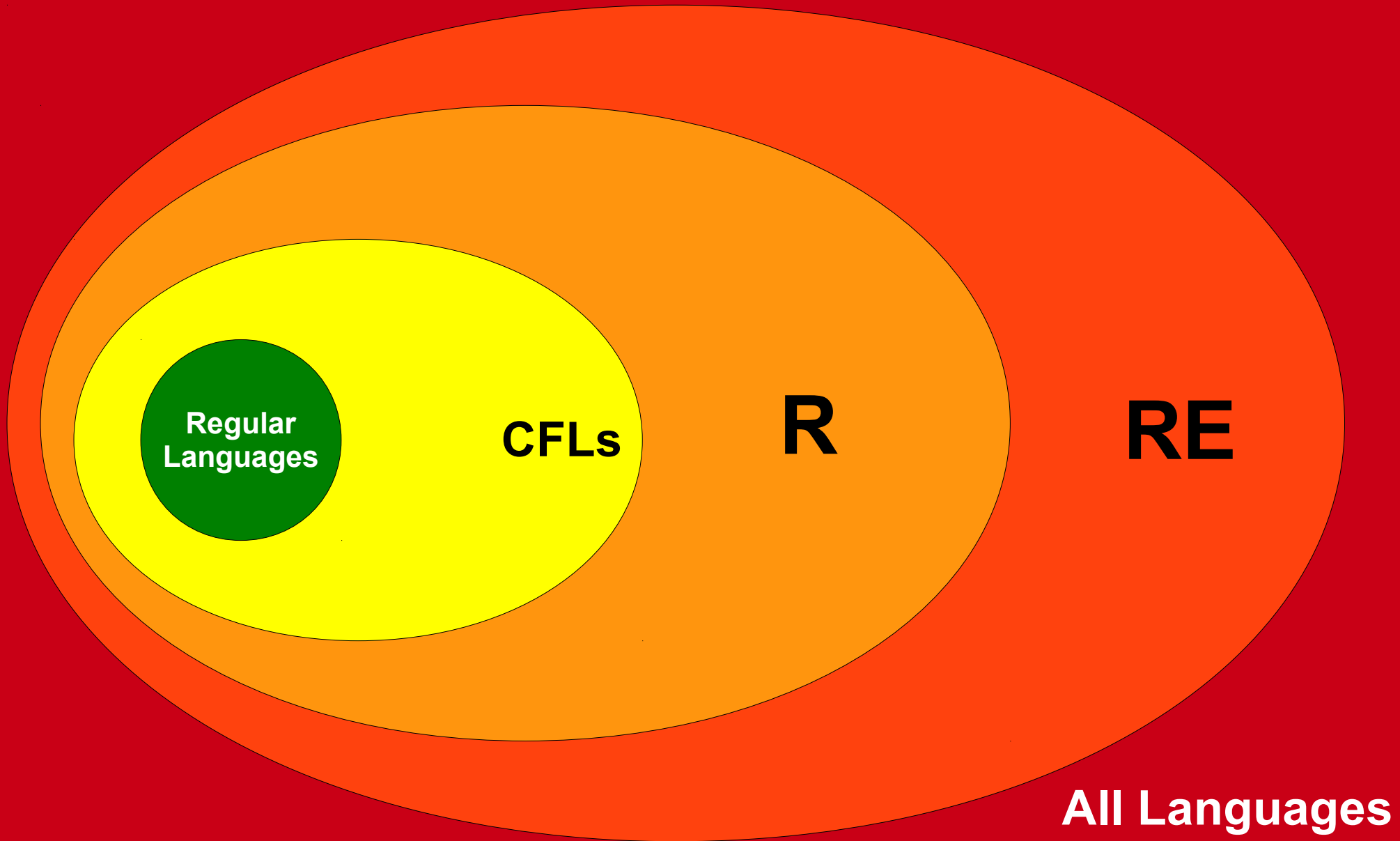
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- 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?




Unanswered Questions

- Why exactly is **RE** an interesting class of problems?
- What does the **$R \stackrel{?}{=} RE$** question mean?
- Is **$R = RE$** ?
- What lies beyond **R** and **RE**?
- We'll see the answers to each of these in due time.

What **problems** can we solve with a computer?

What is a
"problem?"



Decision Problems

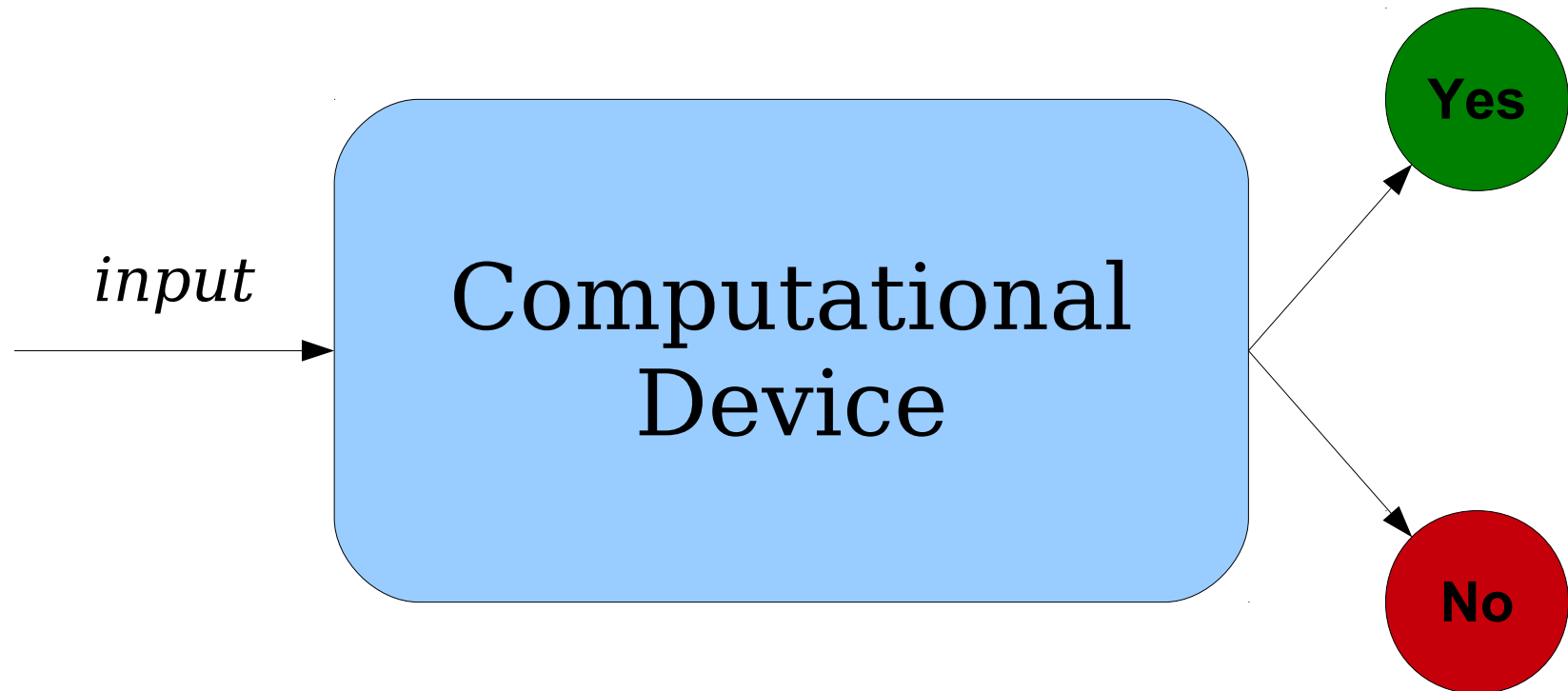
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Bin Packing

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

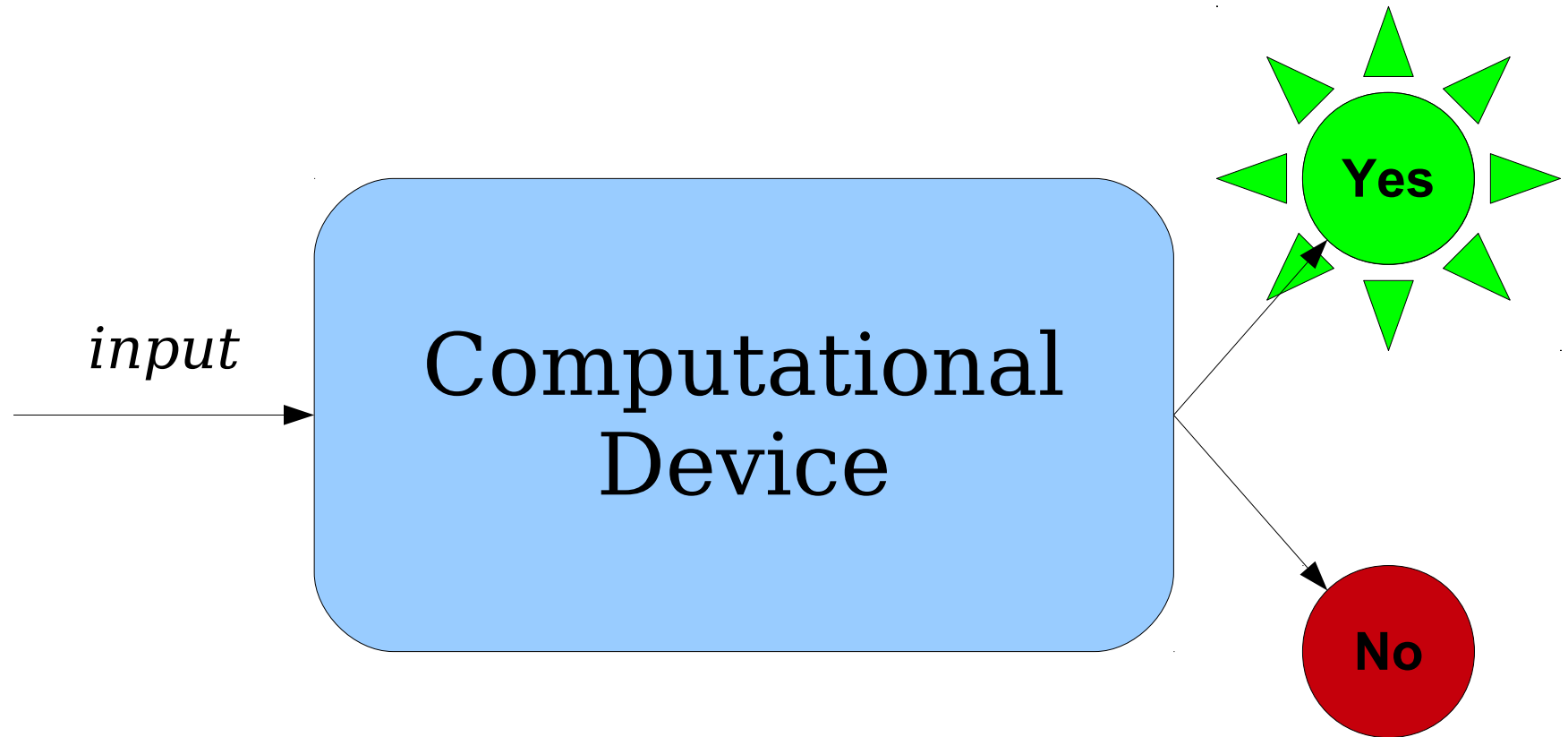
You're given a transportation grid and a number k . Is there a way to place emergency supplies in at most k cities so that every city either has emergency supplies or is adjacent to a city that has emergency supplies?
- Example: Route Planning

You're given the transportation grid of a city, a start location, a destination location, and information about the traffic over the course of the day. Given a time limit T , is there a way to drive from the start location to the end location in at most T hours?

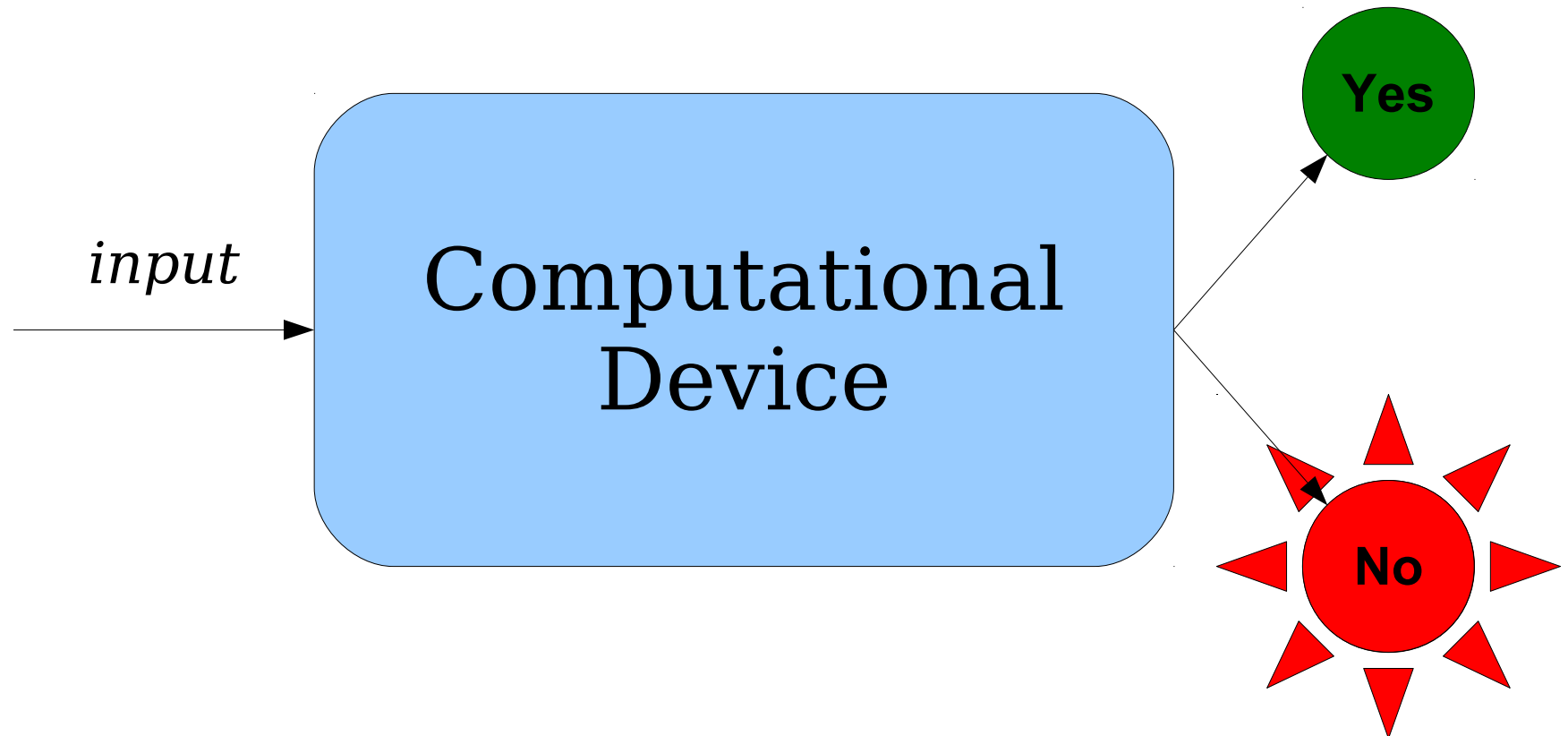
A Model for Solving Problems



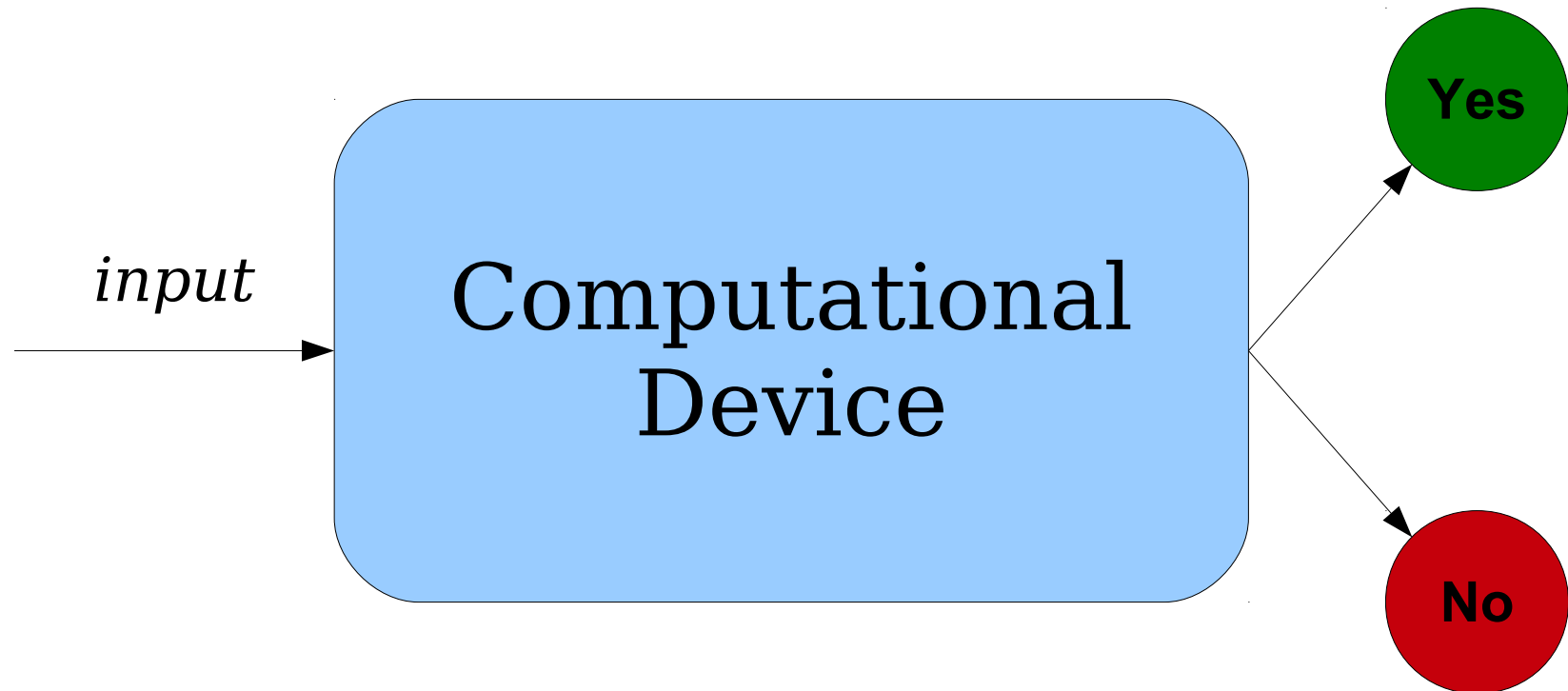
A Model for Solving Problems



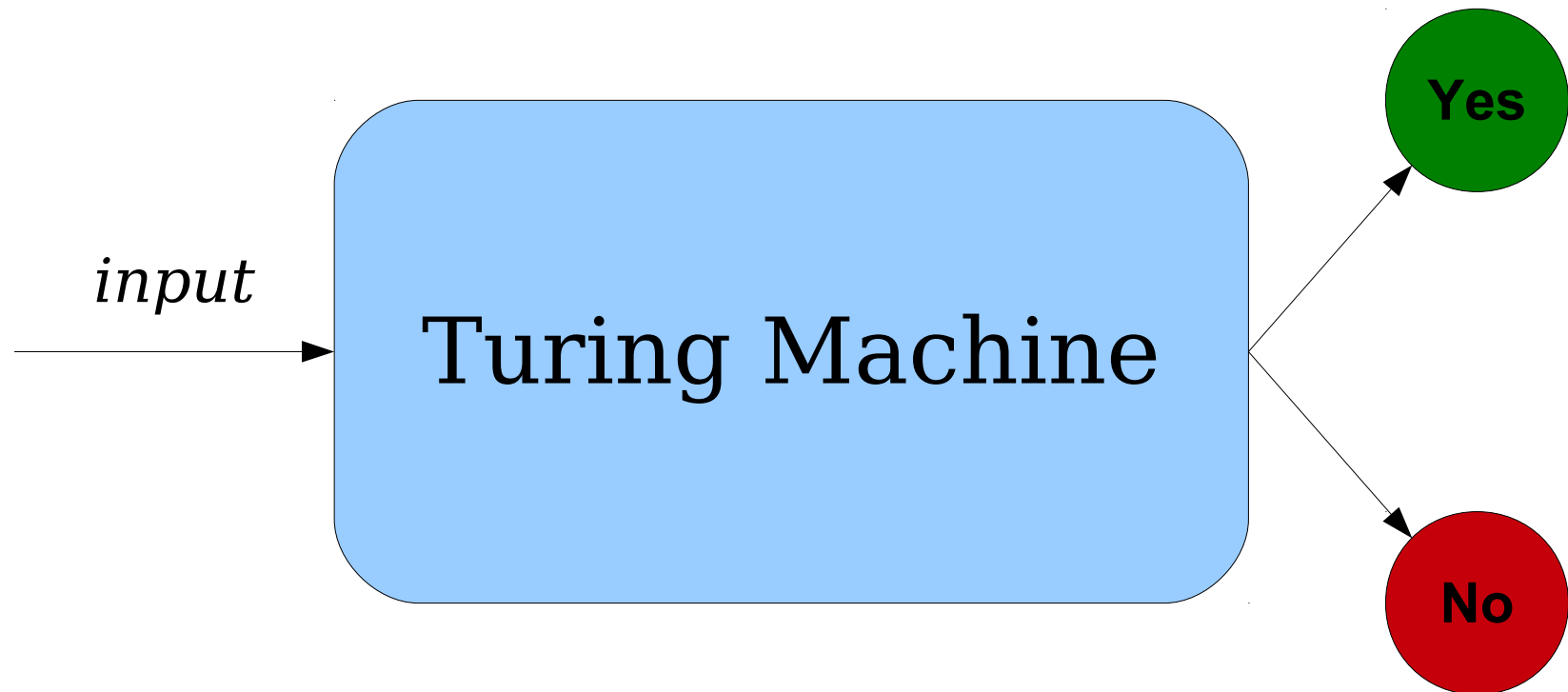
A Model for Solving Problems



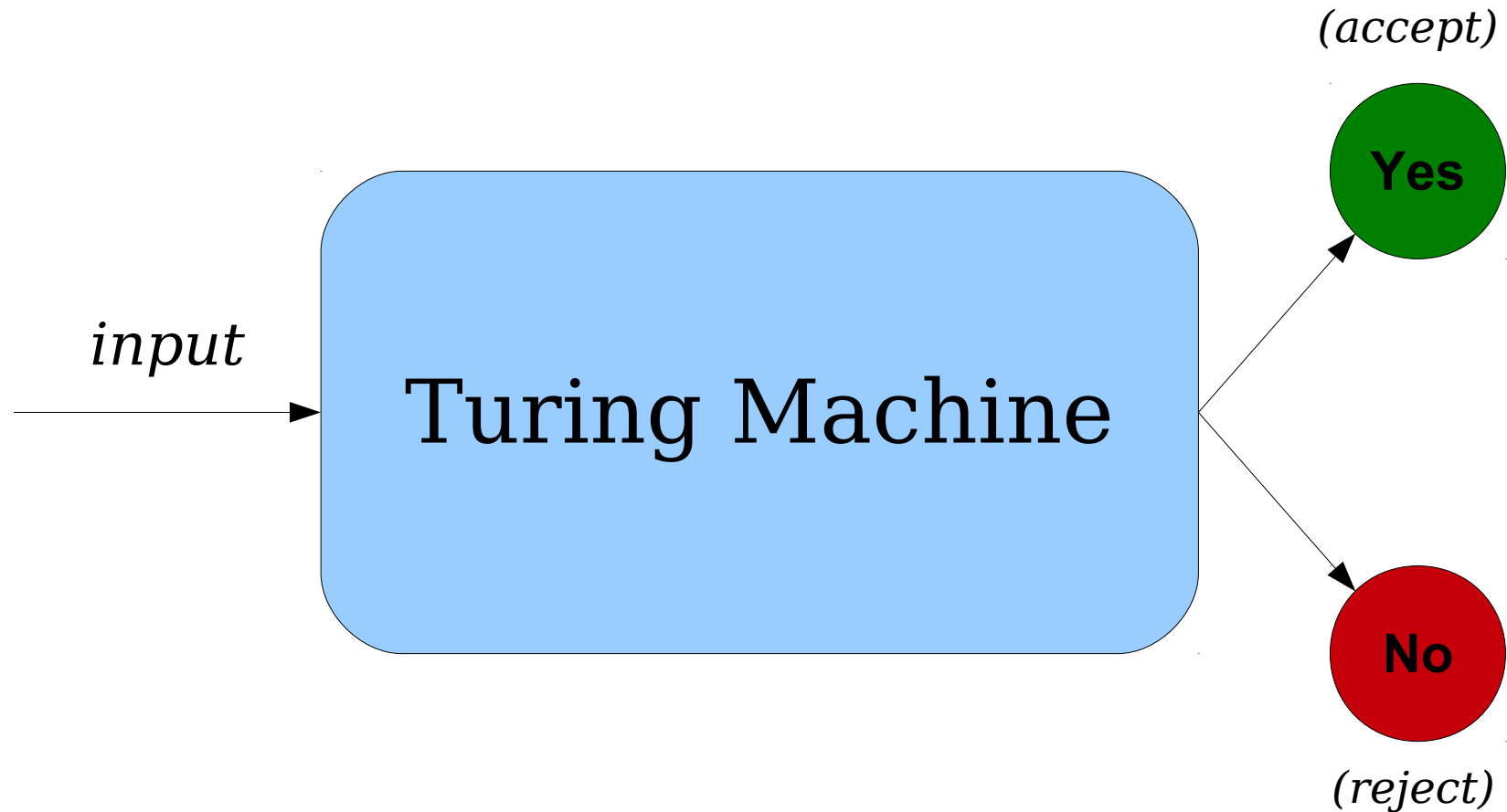
A Model for Solving Problems



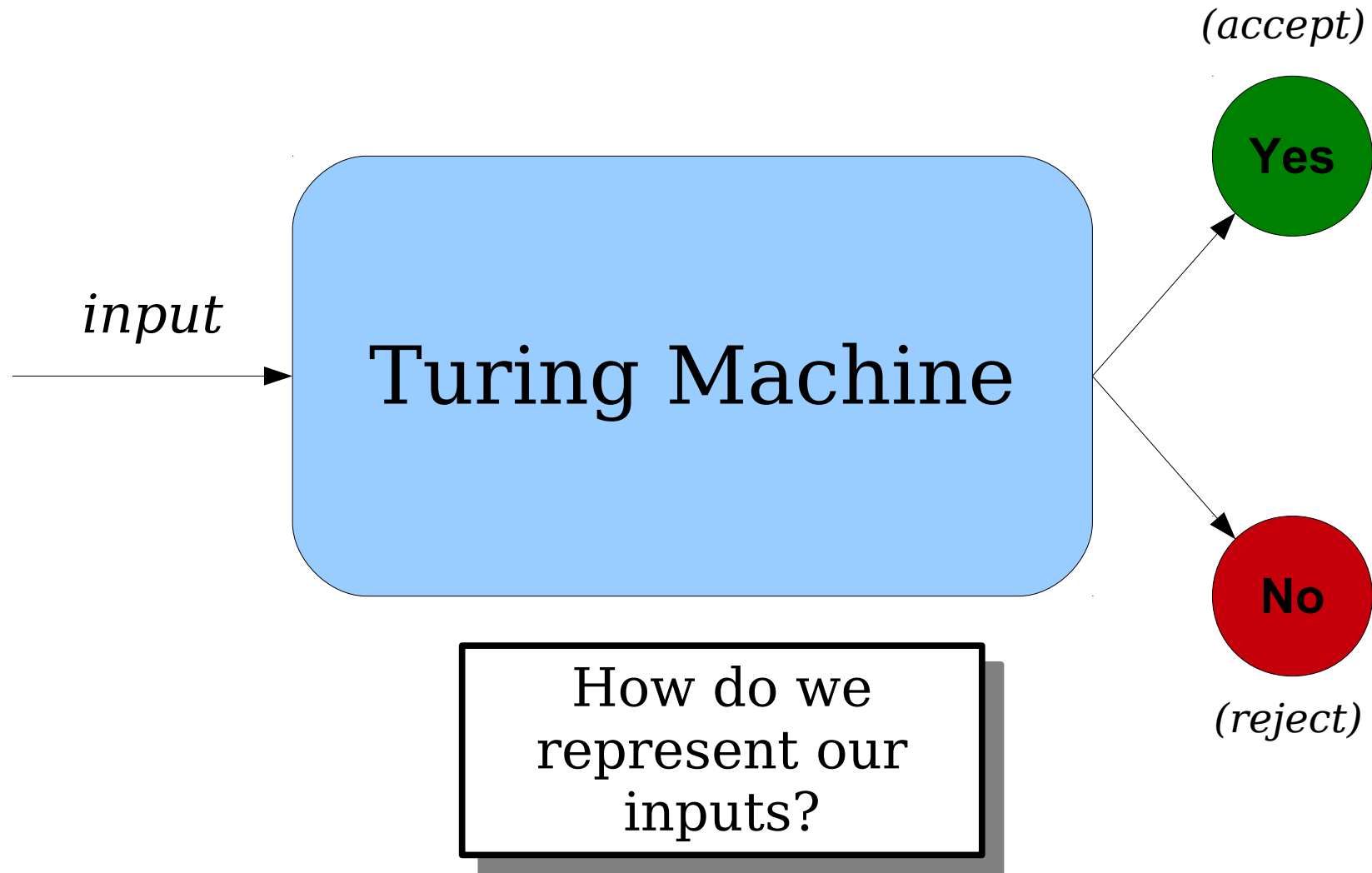
A Model for Solving Problems





A Model for Solving Problems



A Model for Solving Problems



On your computer, *everything* is numbers!

- Images (gif, jpg, png): binary numbers
- Integers (int): binary numbers
- Non-integer real numbers (double): binary numbers
- Letters and words (ASCII, Unicode): binary numbers
- Music (mp3): binary numbers
- Movies (streaming)  : binary numbers
- Doge pictures  : binary numbers
- Email messages: binary numbers

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.
- All data on my computer can be thought of as (suitably-encoded) strings of 0s and 1s.



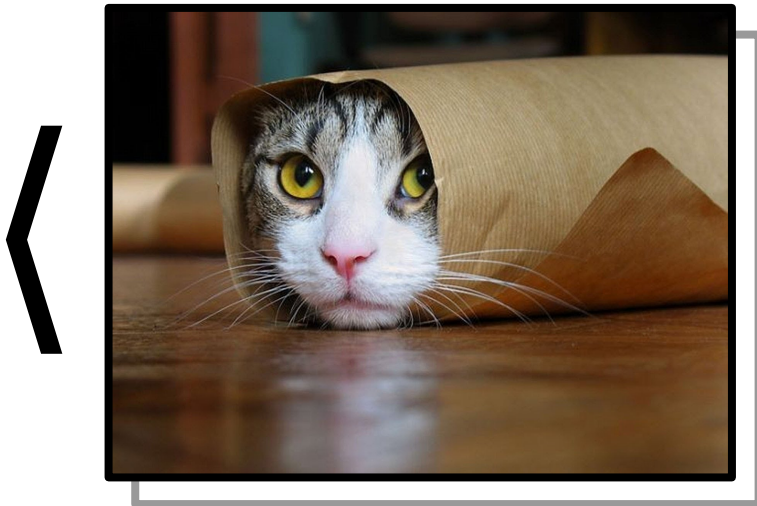
Strings and Objects

- 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 $\langle Obj \rangle$ to refer to some way of encoding that object as a string.
- Think of $\langle Obj \rangle$ like a file on disk – it encodes some high-level object as a series of characters.



= 11011100101110111100010011...110

Object Encodings

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



$\langle Obj \rangle = 00110101000101000101000100\dots001$

Example: Numbers

- Each of the following denotes one way to write the number 24:
 - 24 (decimal)
 - XXIV (Roman numerals)
 - 18 (hexadecimal)
 - 11000 (binary)
 - 𠄎𠄎𠄎𠄎 |||| (tally marks)
 - 二十四 (Chinese numerals)
 - ט"ד (Hebrew numerals)
 - ٢٤ (Arabic numerals)
- Computers are powerful enough to convert any of these formats into any of these other formats. In a sense, what matters more is *what number we're working* with rather than *how that number is represented*.

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 $\langle 137 \rangle$ 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?
- 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.

Non-Examples

- There's no general way to encode real numbers as strings.
 - Imagine a real number generated by tossing infinitely many coins, one for each digit. Heads means "0," tails means "1."
- There's no general way to encode languages as strings.
 - Imagine tossing a coin for each string. Include strings where the coin toss shows heads, leave out strings where the coin toss shows tails.
- **Good test:** If you can figure out a general way to describe a group of objects perfectly precisely in a text file, then you can encode them as strings. If you can't, there's no way to do this.

Encoding Groups of Objects

- Given a group of objects $Obj_1, Obj_2, \dots, Obj_n$, 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 **$\langle Obj_1, \dots, Obj_n \rangle$** .
- This lets us feed multiple inputs into our computational device at the same time.

How many of the following classes of objects can you build string encodings for? (Remember that you need to be able to encode arbitrary objects of each type!)

Binary relations over $\{1, 2, 3, 4\}$

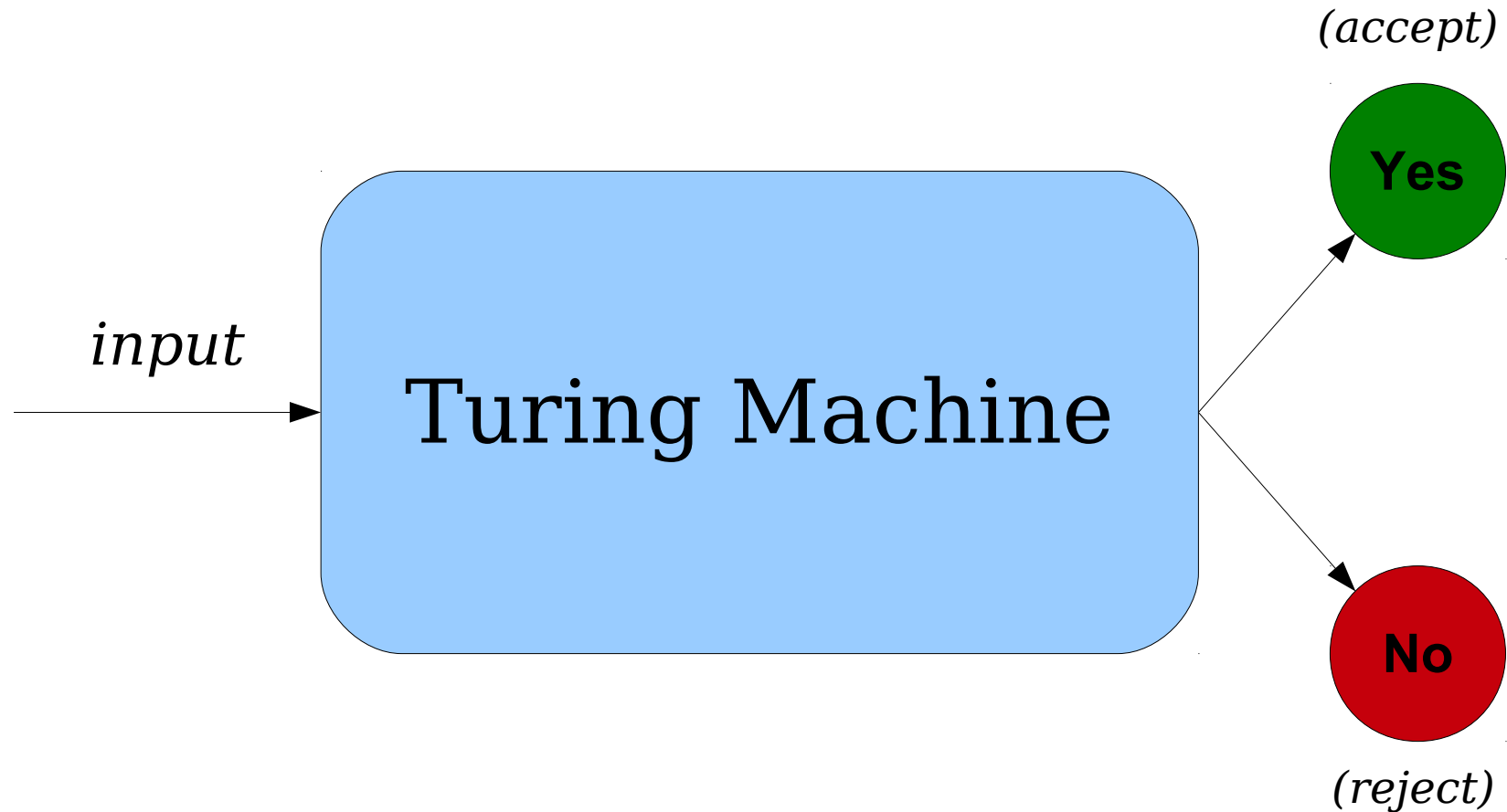
DFAs over $\{\mathbf{a}, \mathbf{b}\}$

Turing machines over $\{\mathbf{a}, \mathbf{b}\}$

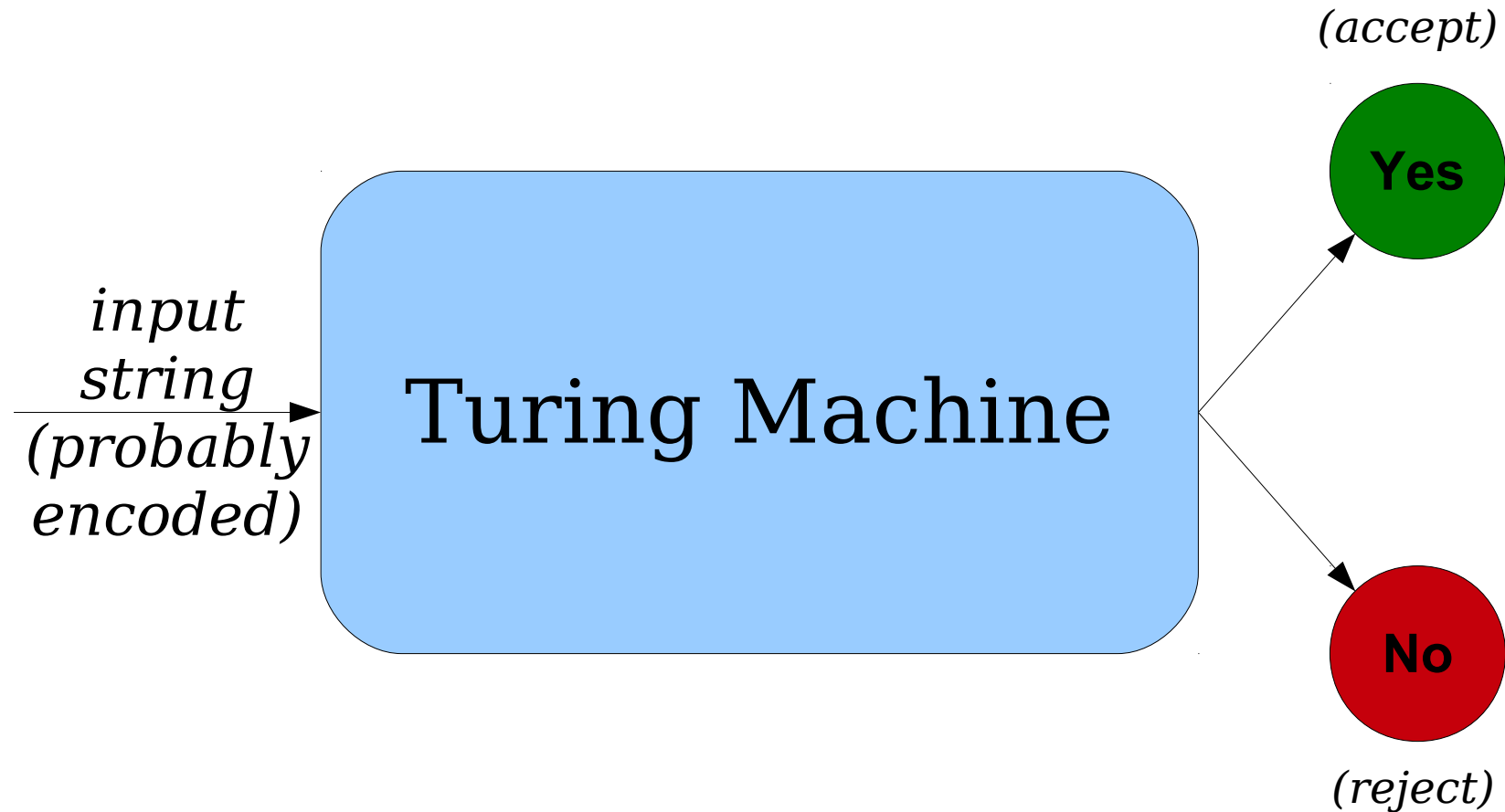
Sets of natural numbers

Answer at [PolleEv.com/cs103](https://www.pollevery.com/cs103) or
text **CS103** to **22333** once to join, then **a number**.

A Model for Solving Problems



A Model for Solving Problems



What All This Means

- Our goal is to speak of *computers solving problems*.
- We will model this by looking at *TMs recognizing languages*.
- For *decision problems* that we're interested in solving, this precisely captures what we're interested in capturing.

Other Models

- Rather than talking about decision problems, we could talk about *function problems*, where we take in an input and produce some output object rather than just a yes/no answer.
- Rather than running a single input through the TM and looking at the result, we could imagine that the TM is constantly running, processing inputs as they arrive.
- These are interesting questions to explore! Take CS154 or CS254 for more details!

What problems can we solve with a computer?

Time-Out for Announcements!

Stanford Math Department
de Leeuw Distinguished Lecture Series
Presents:

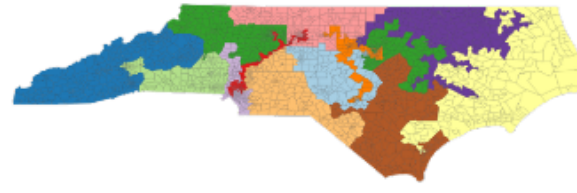
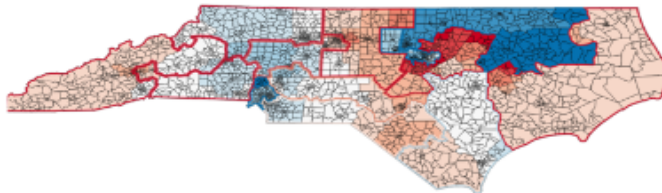


Jonathan Mattingly
(Duke University)

Thursday, March 8, 2018
at 4:30 pm in Building 200-002

Quantifying Gerrymandering: a mathematician goes to court

In October 2017, I found myself testifying for hours in a federal court. I had not been arrested. Rather, I was attempting to quantify gerrymandering using analysis which grew from asking whether a surprising 2012 election was indeed surprising. It hinged on probing the geopolitical structure of North Carolina using a Markov Chain Monte Carlo algorithm. I will start at the beginning and describe the mathematical ideas involved in our analysis. The talk will be accessible to undergraduates. In fact, this project began as a sequence of undergraduate research projects and undergraduates continue to be involved to this day.



Problem Set Eight

- Problem Set Eight is due this Friday at 2:30PM.
 - This is the last problem set that you can use late days on... but you should be strategic with doing so because PS9 will be due next Friday at 2:30PM sharp.
- As always, come talk to us if you need help! Ask questions on Piazza or in office hours.

Back to CS103!

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 thought and consciousness.
 - 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. You can't have computation without these properties.
- 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?

Computers and Programs

- When talking about actual computers, most people just have a single computer.
- To get the computer to perform a particular task, we load a program into it and have the computer execute that program.
- In certain cases it's faster or more efficient to make dedicated hardware to solve a problem, but the benefits of having one single computer outweigh the costs.
- **Question:** Can we do something like this for Turing machines?

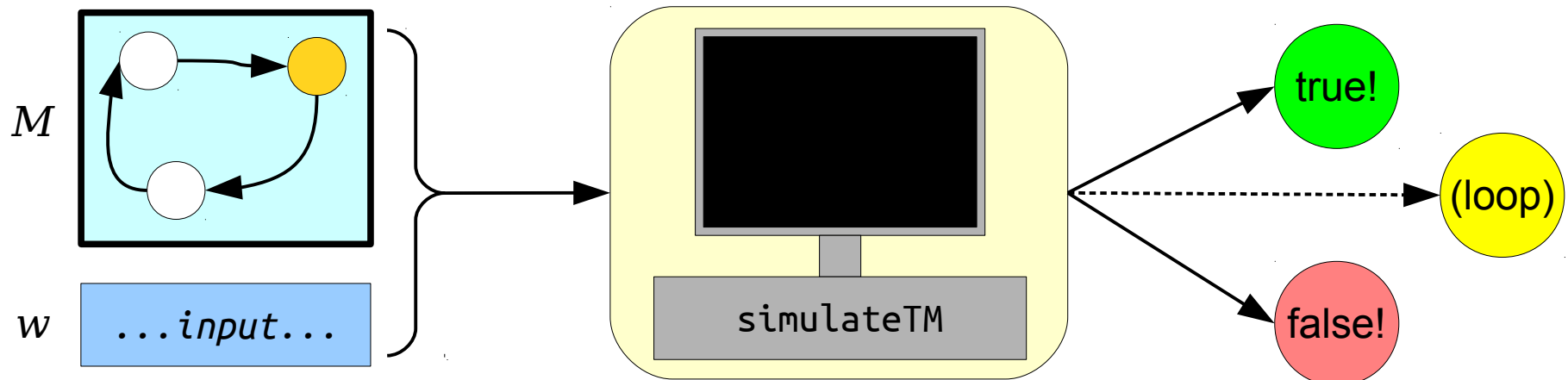
A TM Simulator

- 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*)

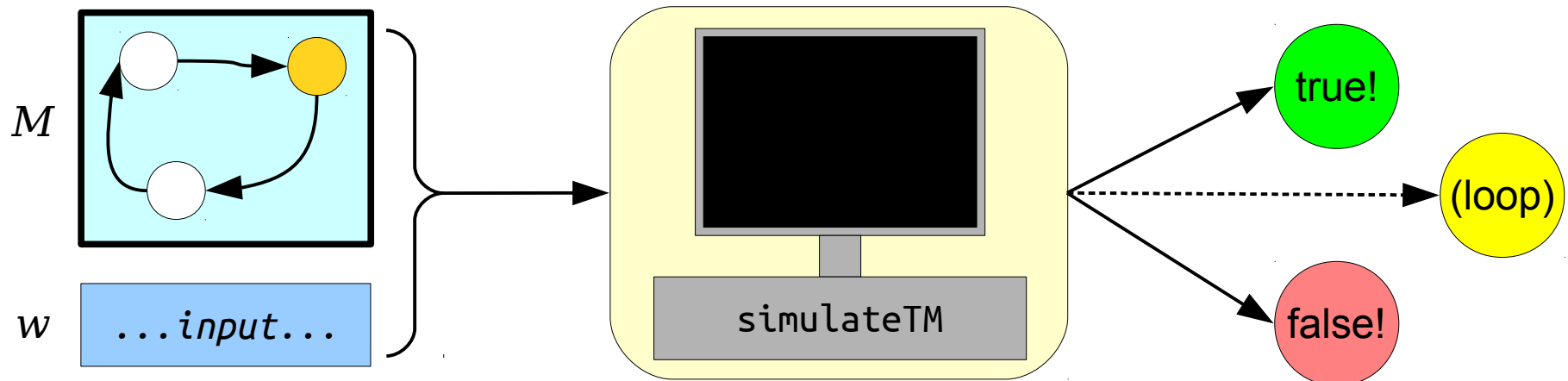
with the following behavior:

- 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.



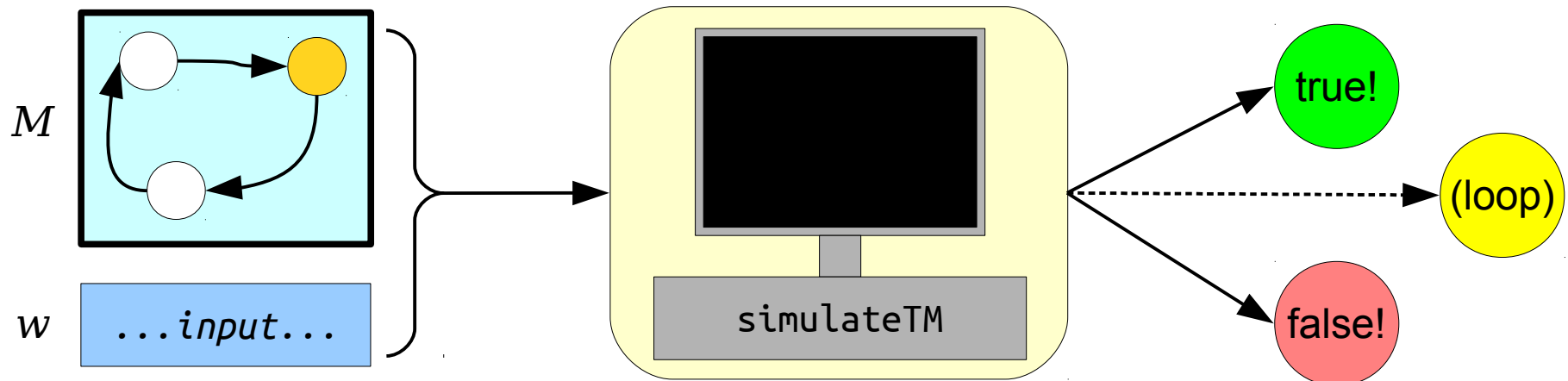
A TM Simulator

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



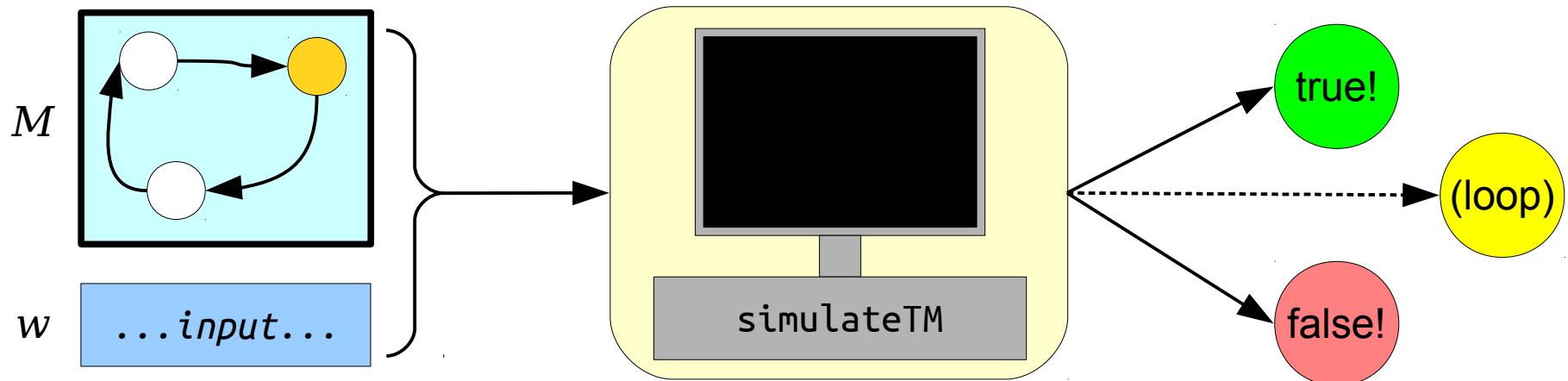
A TM Simulator

- 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.



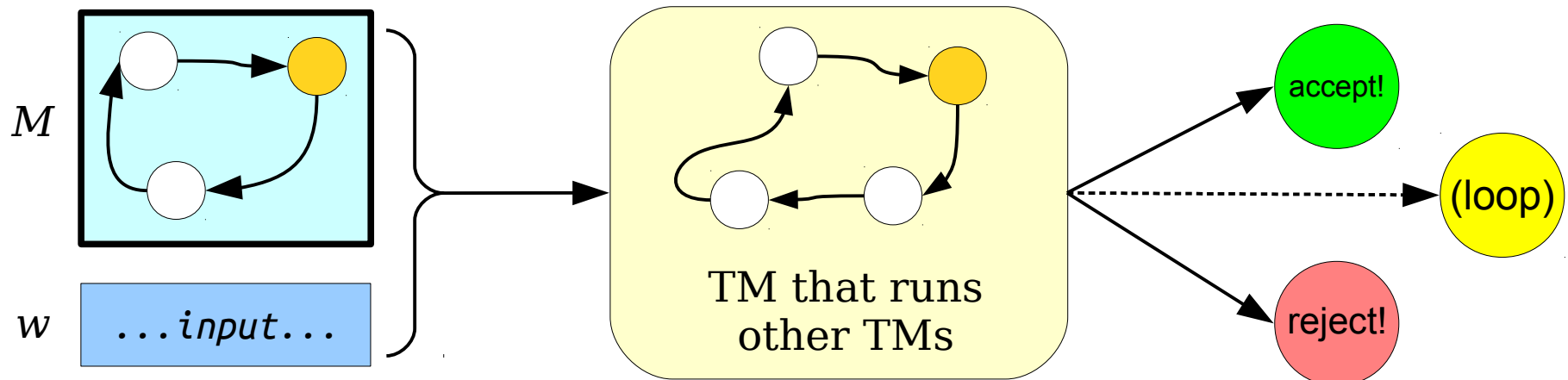
A TM Simulator

- 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?



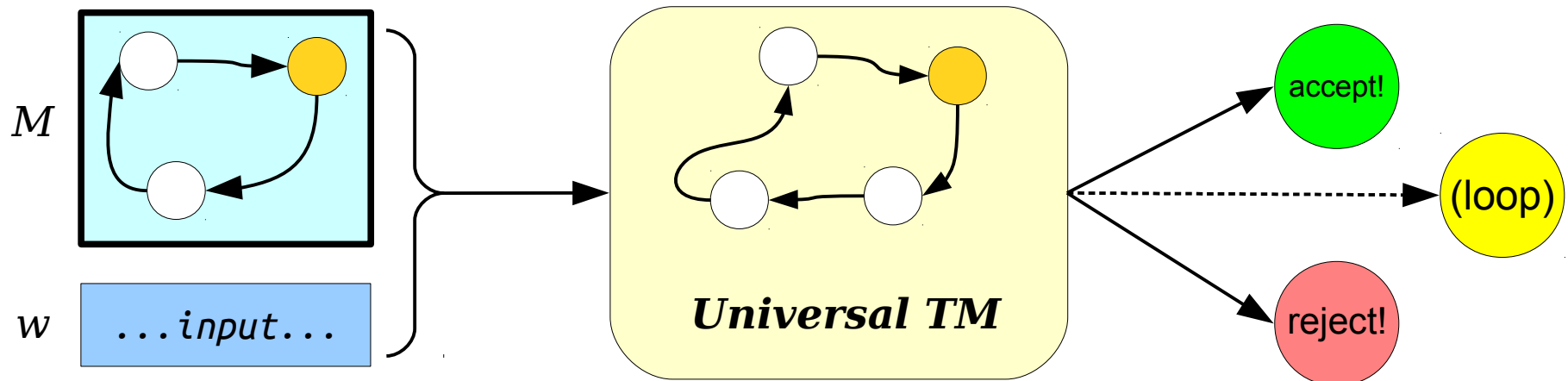
A TM Simulator

- 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?



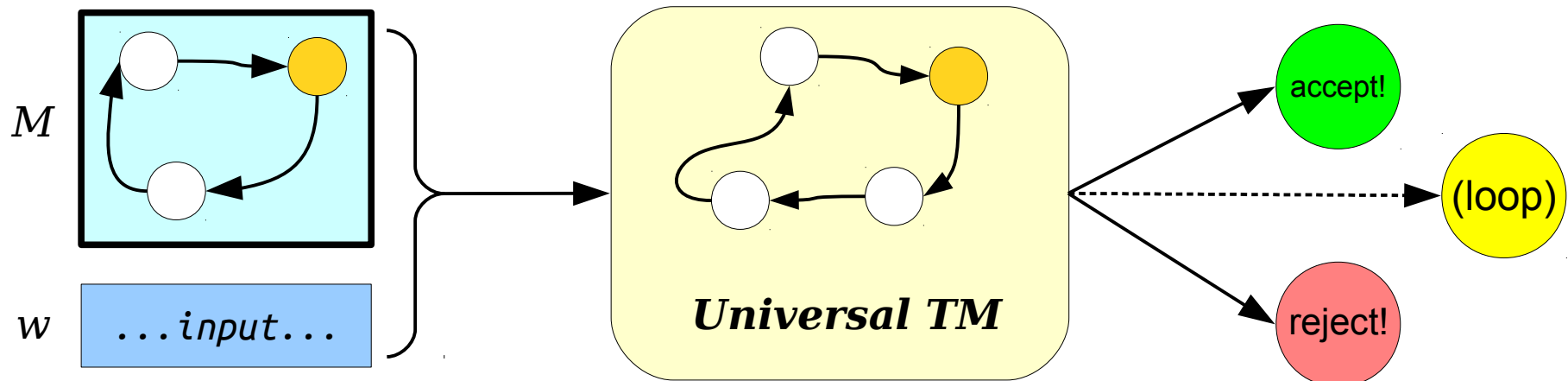
A TM Simulator

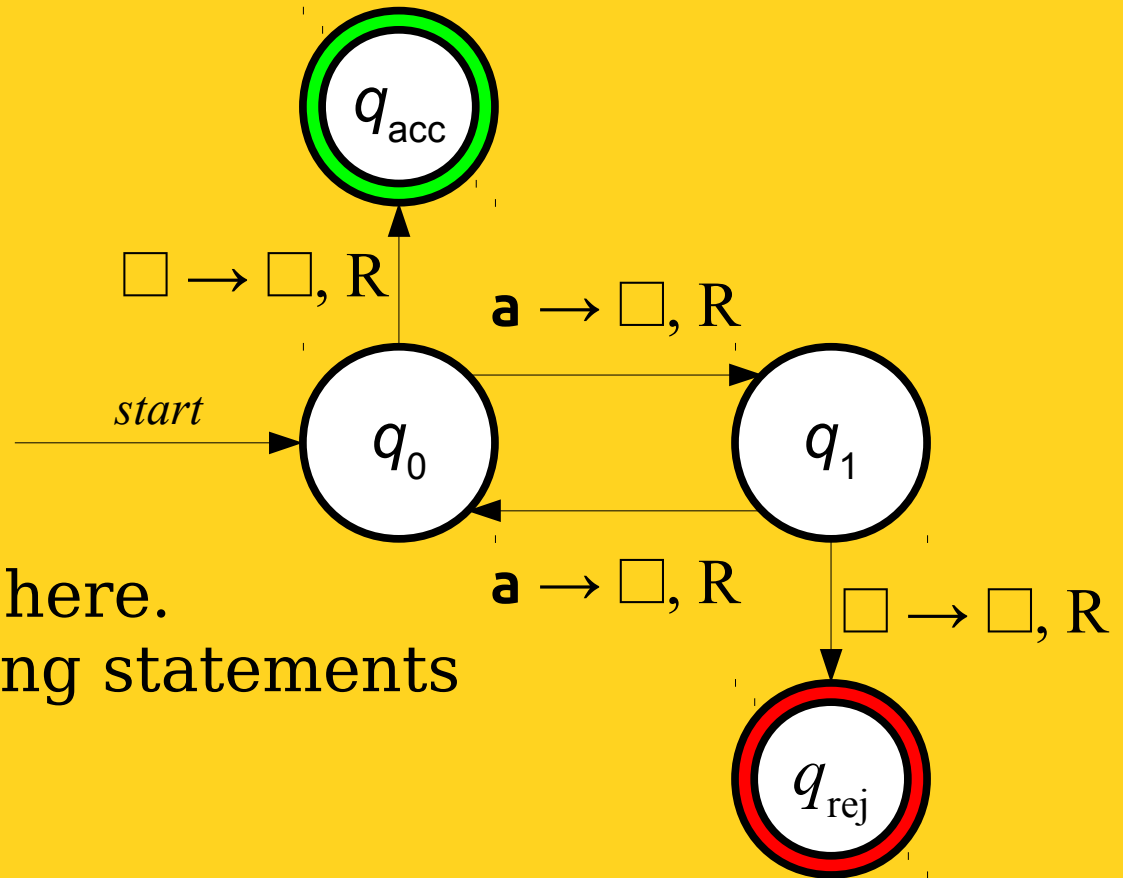
- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



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 $\langle M, w \rangle$.
- **U_{TM} accepts $\langle M, w \rangle$ if and only if M accepts w .**





Let M be the TM shown here.
 How many of the following statements
 are true?

M accepts **aa**

U_{TM} accepts $\langle M, \mathbf{aa} \rangle$

U_{TM} accepts $\langle U_{TM}, \langle M, \mathbf{aa} \rangle \rangle$

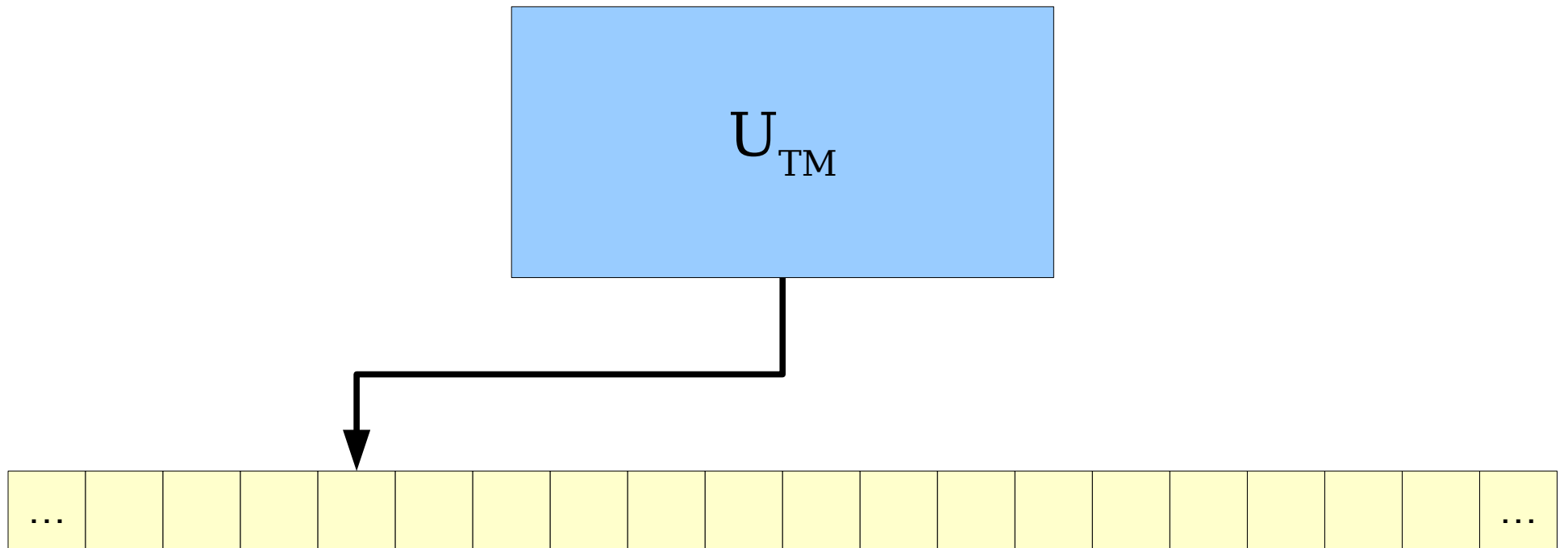
U_{TM} accepts $\langle U_{TM}, \langle U_{TM}, \langle M, \mathbf{aa} \rangle \rangle \rangle$

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

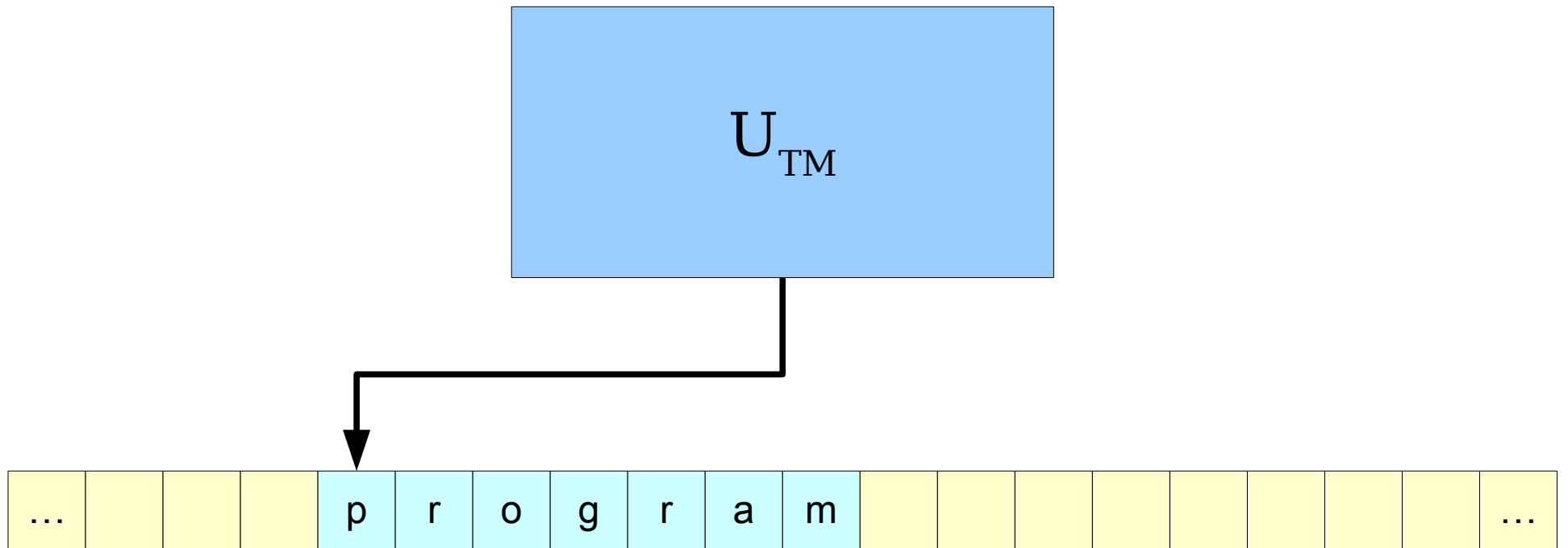
An Intuition for U_{TM}

- You can think of U_{TM} as a general-purpose, programmable computer.
- Rather than purchasing one TM for each language, just purchase U_{TM} and program in the “software” corresponding to the TM you actually want.
- U_{TM} is a powerful machine: ***it can perform any computation that could be performed by any feasible computing device!***

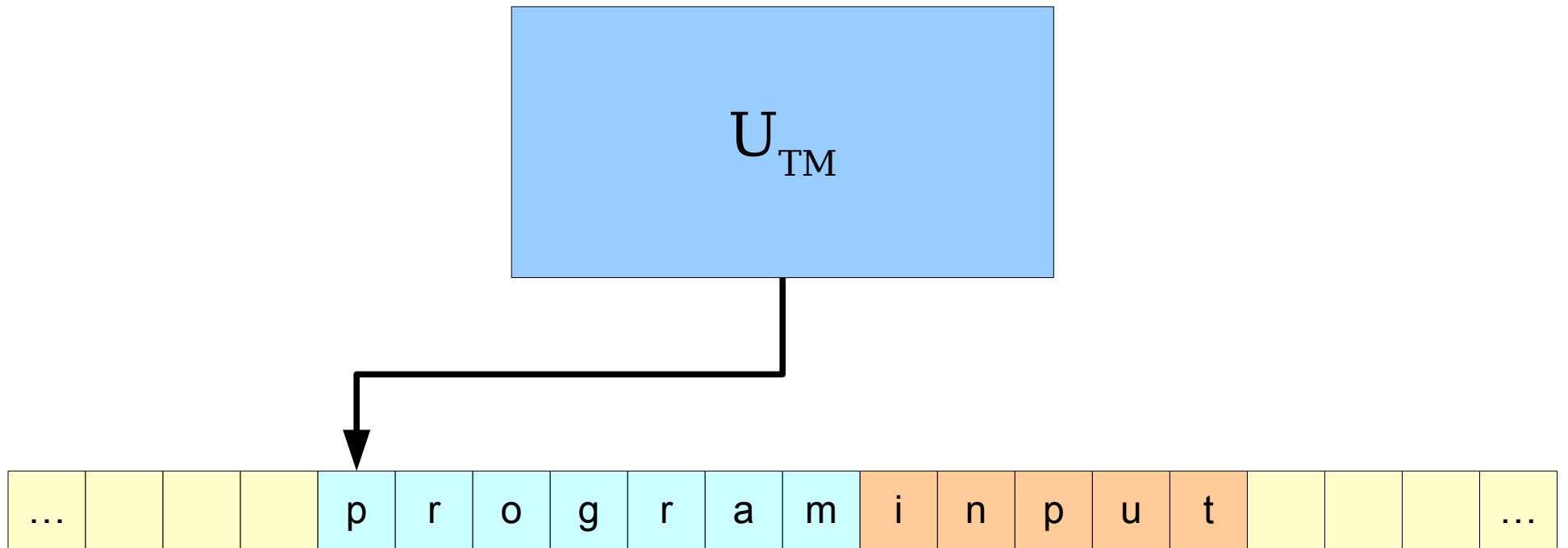
A Universal Machine



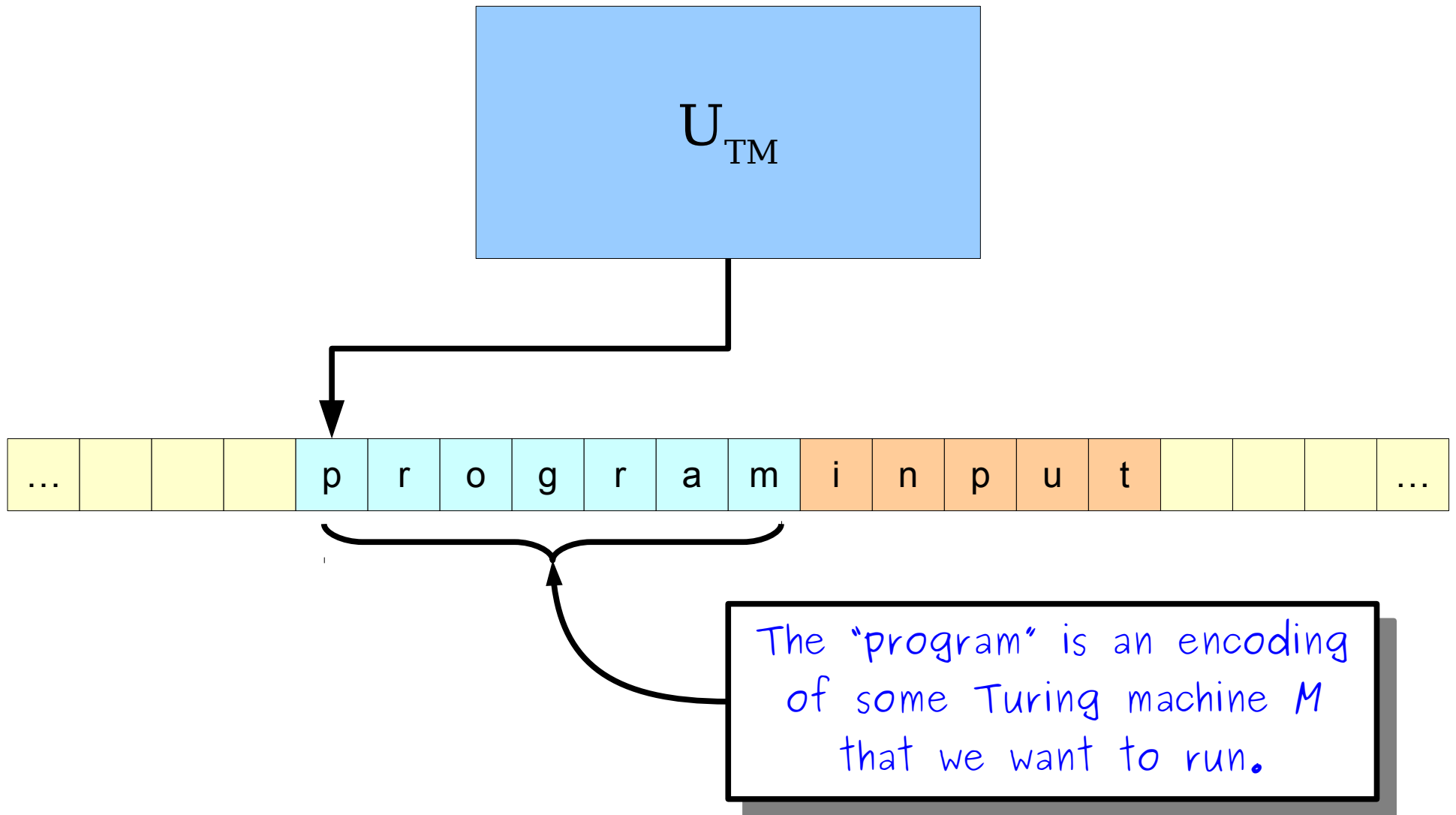
A Universal Machine



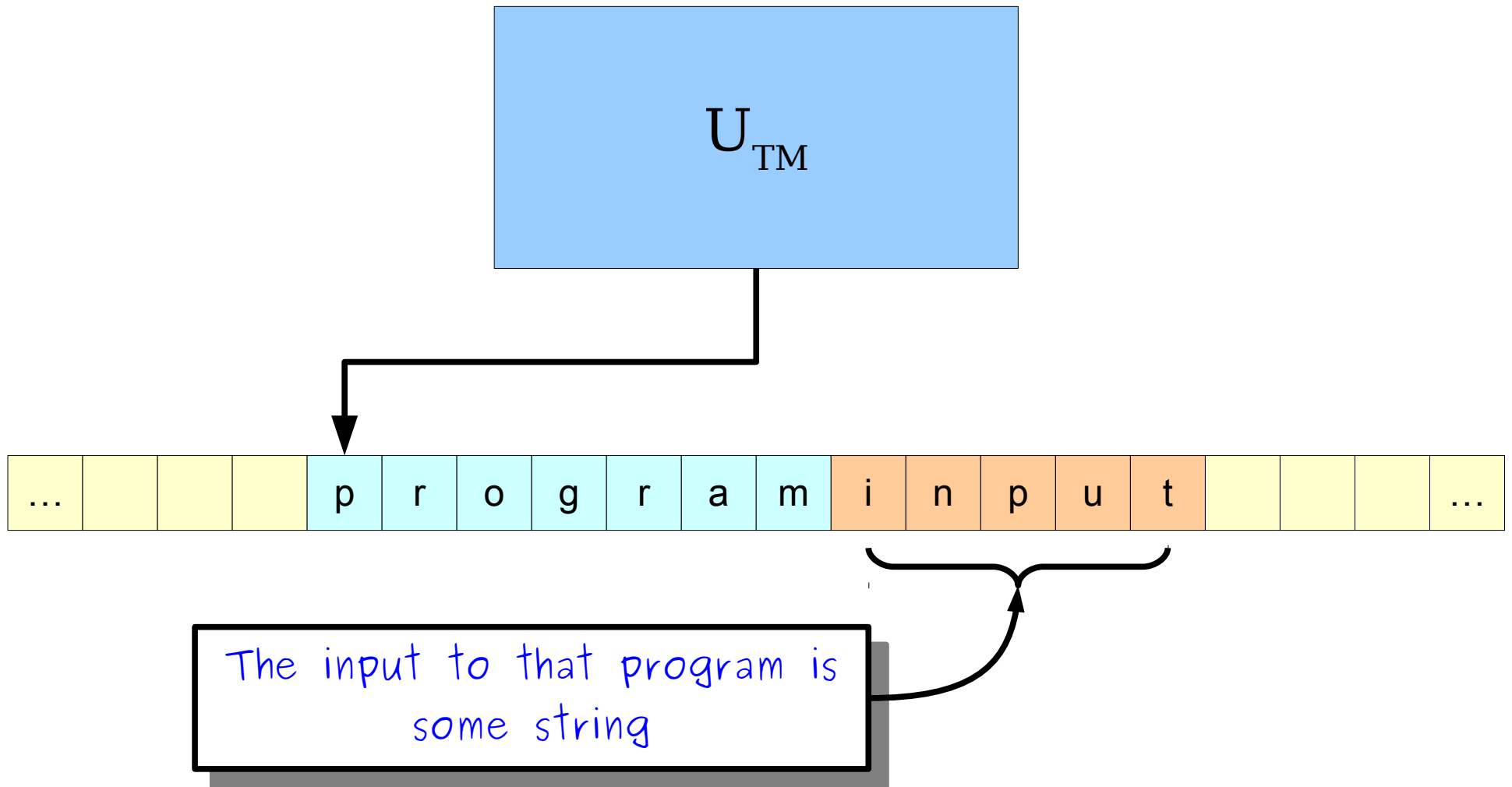
A Universal Machine



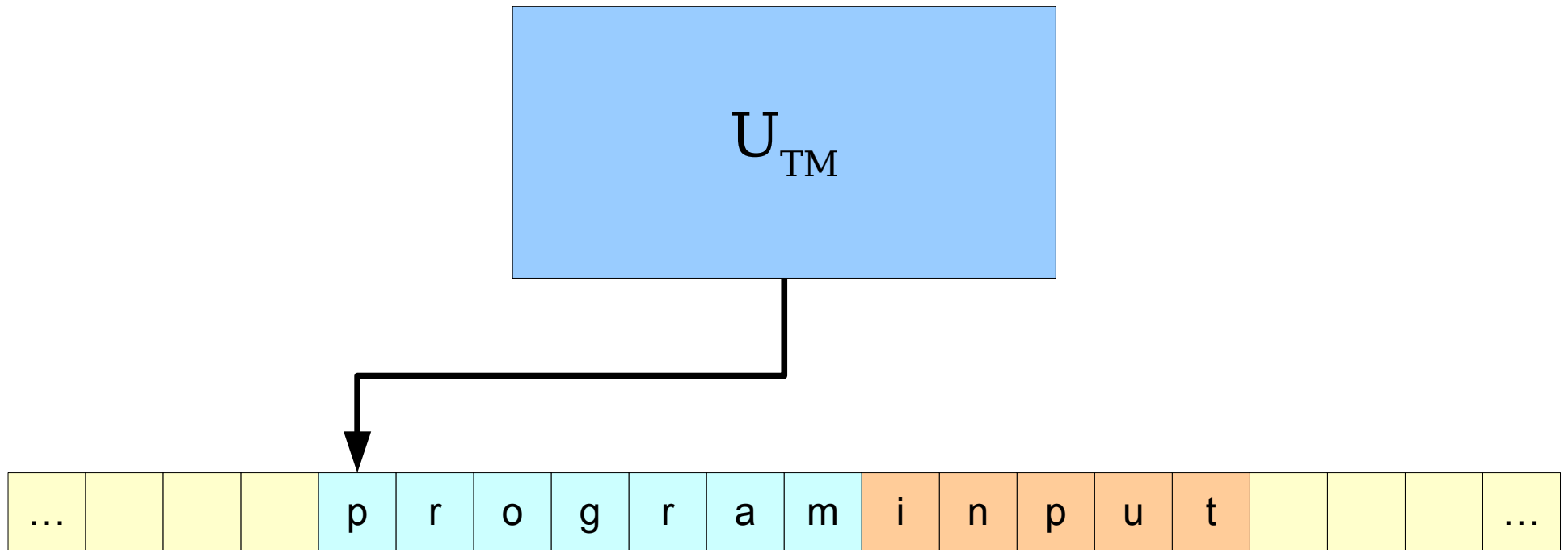
A Universal Machine



A Universal Machine

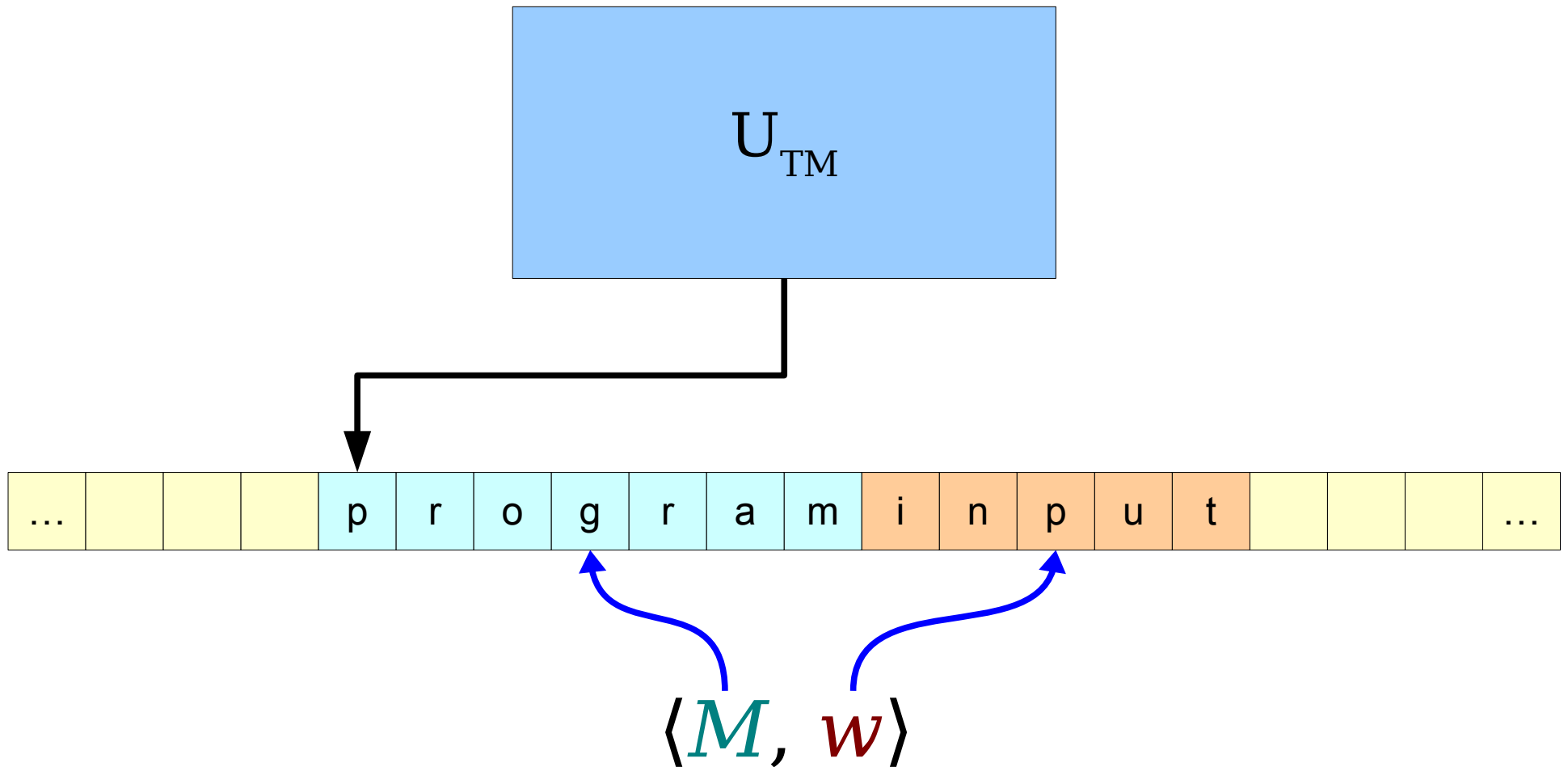


A Universal Machine



The input has the form $\langle M, w \rangle$, where M is some TM and w is some string.

A Universal Machine



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: For any TM M , the language of M , denoted $\mathcal{L}(M)$, is the set

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

- What is the language of U_{TM} ?
- U_{TM} accepts $\langle M, w \rangle$ iff M is a TM that accepts w .
- Therefore:

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

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

- For simplicity, define $A_{\text{TM}} = \mathcal{L}(U_{\text{TM}})$. This is an important language and we'll see it many times.

$$\begin{aligned} A_{\text{TM}} &= \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \\ &= \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M) \} \end{aligned}$$

Let M be a TM where $\mathcal{L}(M) = \{ \mathbf{a}^n \mathbf{b}^n \mid n \in \mathbb{N} \}$

How many of the following statements are true?

$$\langle M, \varepsilon \rangle \in A_{\text{TM}}$$

$$\langle M, \mathbf{a} \rangle \in A_{\text{TM}}$$

$$\langle M, \mathbf{b} \rangle \in A_{\text{TM}}$$

$$\langle M, \mathbf{ab} \rangle \in A_{\text{TM}}$$

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

Regular Languages

CFLs



A_{TM}

RE

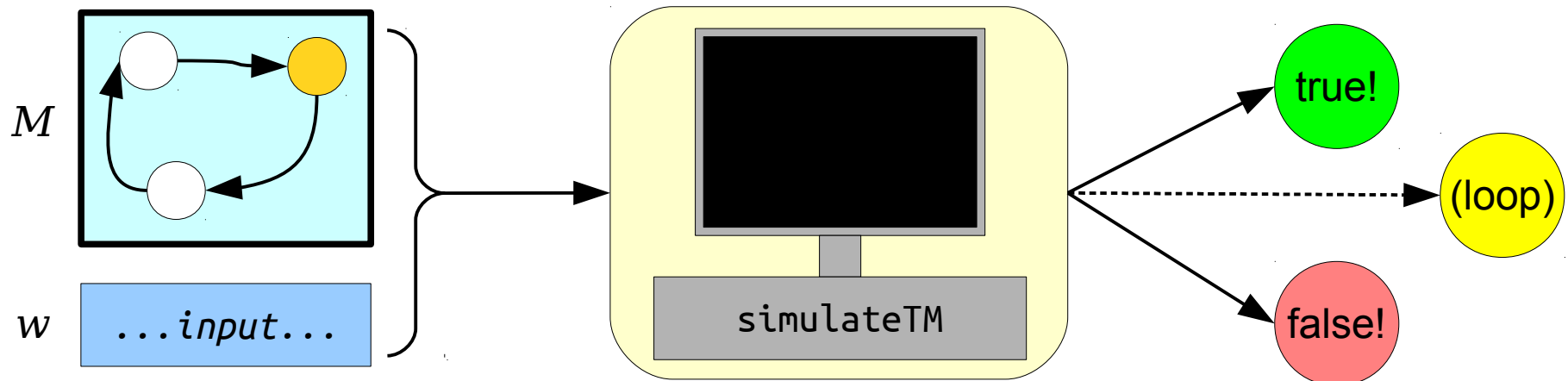
All Languages

Uh... so what?

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

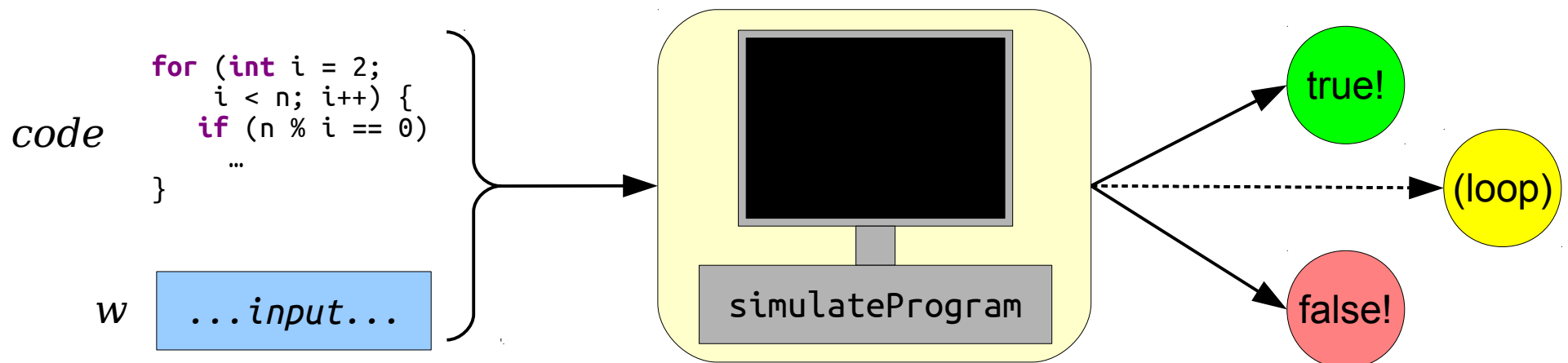
Why Does This Matter?

- 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?