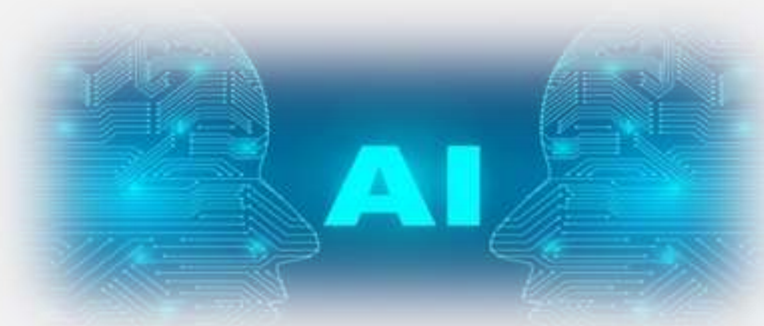
Church-Turing Thesis



The Church-Turing Thesis

Alonzo Church (1903 – 1995) was an American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science. Turing was his student at Princeton.

- How powerful are TMs?
- What problems can you solve with a computer?
- What does it mean to solve a problem?
  - Rethinking what "solving" a problem means, and two possible answers to that question

Real and “Ideal” Computers

- A real computer has memory limitations: you have a finite amount of RAM, a finite amount of disk space, etc
- 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.

Theorem

Turing machines are equal in power to idealized computers. That is, any computation that can be done on a TM can be done on an idealized computer and vice-versa.

Key Idea: Two models of computation are equally powerful if they can simulate each other.

Simulating a TM

- The individual commands in a TM are simple and perform only basic operations:

  Move

  Write

  Goto

  Return

  If

- The memory for a TM can be thought of as a string with some number keeping track of the current index
- To simulate a TM, we need to
  – see which line of the program we're on
  – determine what command it is
  – simulate that single command
- Claim: This is reasonably straightforward to do on an idealized computer

Simulating a TM

- Because a computer can simulate each individual TM instruction, a computer can do anything a TM can do

- Key Idea: Even the most complicated TM is made out of individual instructions, and if we can simulate those instructions, we can simulate an arbitrarily complicated TM

Simulating Computer

Simulating a Computer

- Programming languages provide a set of simple constructs
  - Think things like variables, arrays, loops, functions, classes, etc.
- You, the programmer, then combine these basic constructs together to assemble larger programs
- Key Idea: If a TM is powerful enough to simulate each of these individual pieces, it's powerful enough to simulate anything a real computer can do

Can TMs Do: Loops?

- We've seen TMs use loops to solve problems
  - The TM for $L = \{ a^n b^n \mid n \in \mathbb{N} \}$ repeatedly pulls off the first and last character from the string
  - Our sorting TM repeatedly finds ba and replaces it with ab
- In some sense, the existence of Goto and labels means that TMs have loops
- Hopefully, it's not too much of a stretch to think that TMs can do while loops, for loops, etc.

Can TMs Do: Arithmetic?

AI

- TMs can perform basic arithmetic
  - Addition of two numbers
  - We can check if two numbers are equal
- We could also do addition and subtraction, compute powers of numbers, do ceilings and floors, etc

Can TMs Do: Variables?

- TMs can maintain variables
  - You can think of the TM for $L = \{ a^n b^n \mid n \in \mathbb{N} \}$ as storing two variables - one that counts a number of a's, and one that counts a number of b's
  - The TM for Fibonacci numbers tracks the last two Fibonacci numbers, plus the length of the input string

Can TMs Do: Helper Functions?

- We’ve seen TMs with helper functions
  - We saw how to check for equal numbers of a’s and b’s by first sorting the string, then checking of the string has the form $a^n b^n$
  - We can check if a decimal number is a Fibonacci number by converting it to unary, then running our unary Fibonacci checker

- A TM could have multiple “helper functions” that work together to solve some larger problem.

What Else Can TMs Do?

- Maintain strings and arrays
  - Store their elements separated with some special separator character
- Support pointers
  - Maintain an array of what's in memory, where each item is tagged with its "memory address."
- Support function call and return
  - It's hard, but you can do this if you can do helper functions and variables

A TM Can Do What Computers Do

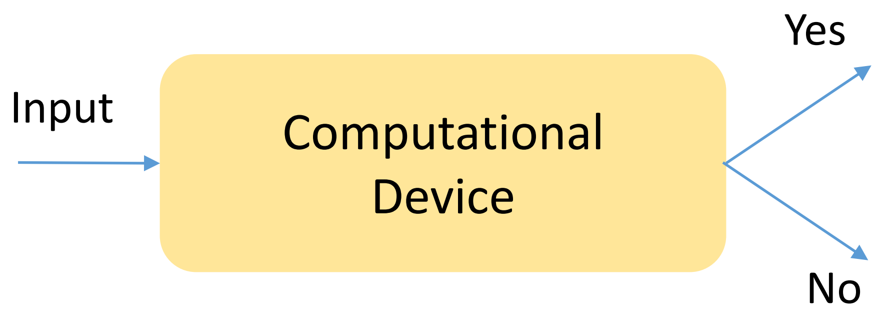
- Internally, computers execute by using basic operations like
  - Simple arithmetic
  - Memory reads and writes
  - Branches and jumps
  - Register operations
  - Etc.

- Each of these are simple enough that they could be simulated by a Turing machine.

TM ≡ Computer

TM ≡ Computer

- Claim: A TM is powerful enough to simulate any computer program that gets an input, processes that input, then returns some result



- 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

TM Can Work with

- Images
  - A picture is just a 2D array of colors, and a color can be represented as a series of numbers
- Video
  - Just a series of pictures
- Music
  - Easy
- AI
  - Symbol manipulation
- Deep Learning
  - That’s just applying a bunch of matrices and nonlinear functions to some input

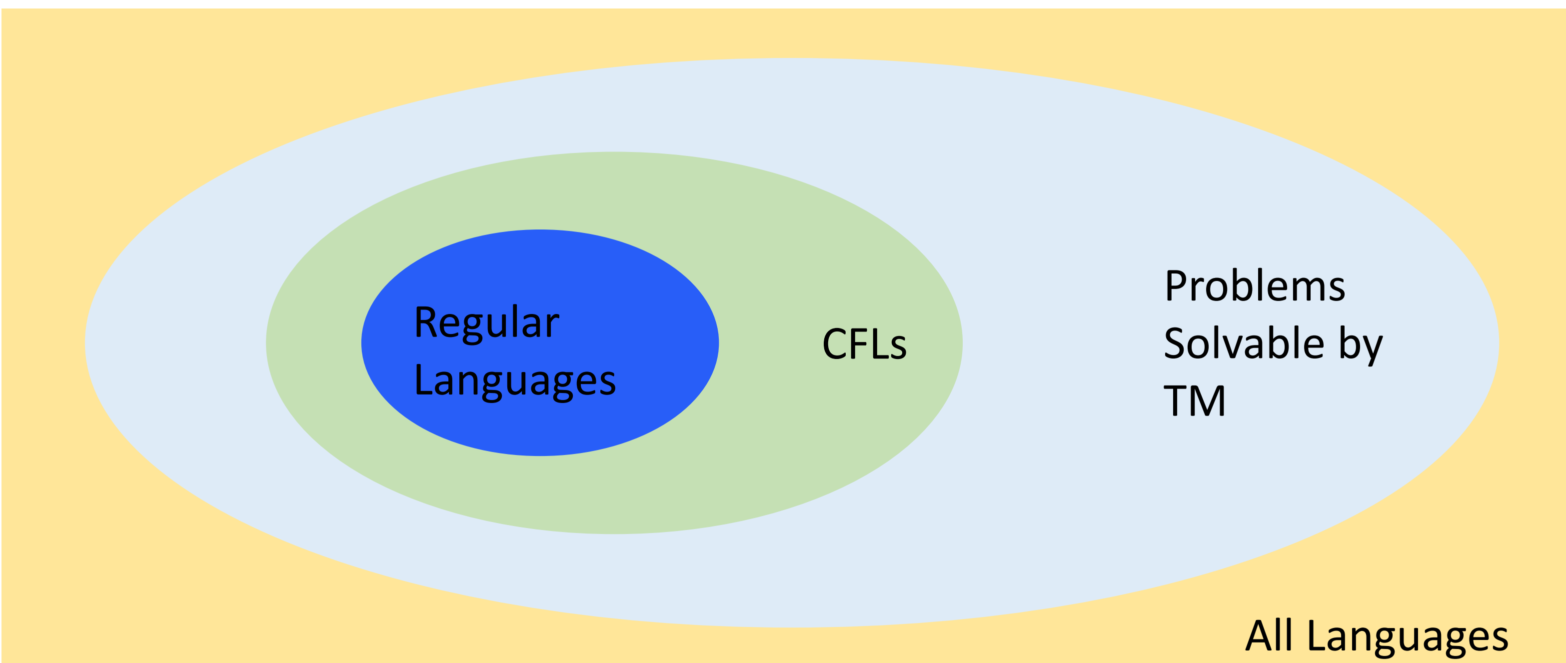
Effective Computation

- 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

Church-Turing Thesis

- 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!”



TMs and Computation

- 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

Finite/Infinite Loops

We'll Learn

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

What does it mean to "solve" a problem?

What kind of computer?

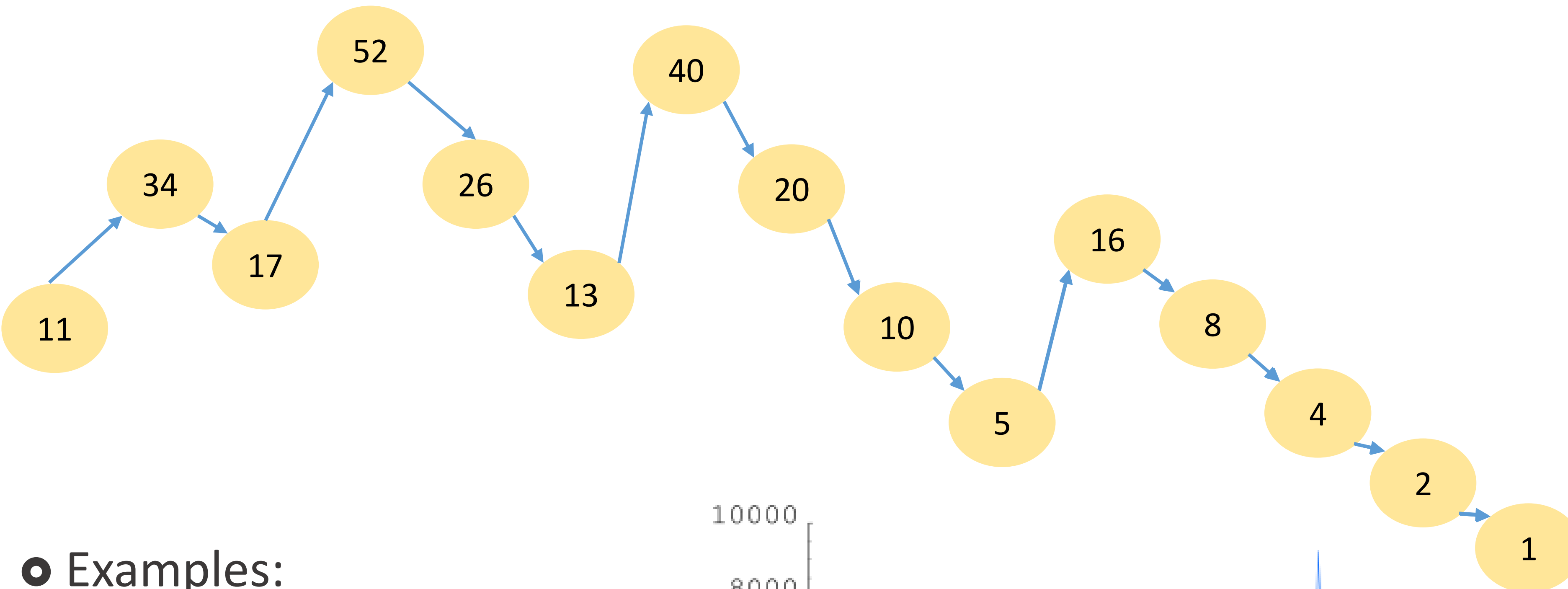
The Hailstone Sequence

- 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

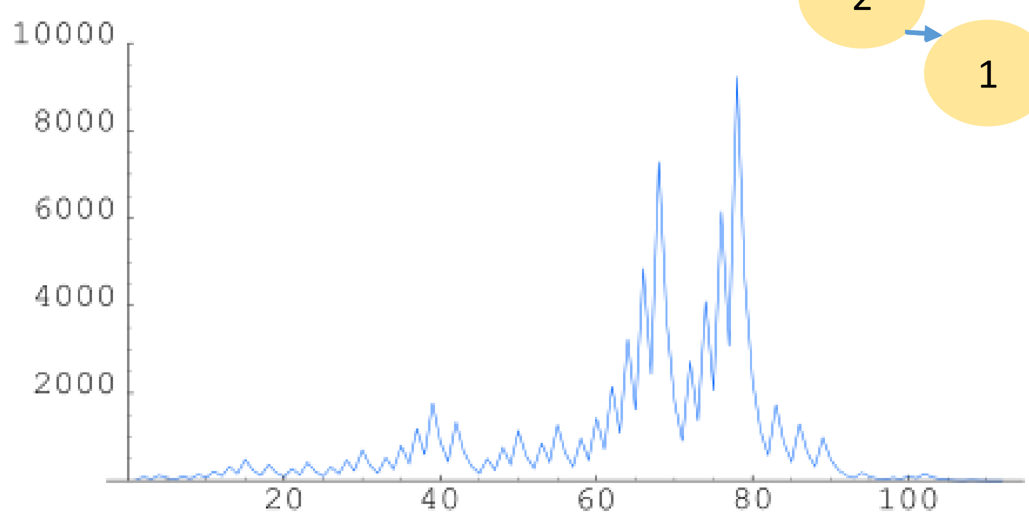
- Question: Given a natural number $n > 0$, does this process terminate?

The Hailstone Sequence

- Example: n= 11



- Examples:
  - n = 5? Yes, after 5 steps
  - n = 20? Yes, after 7 steps
  - n = 7? Yes, after 16 steps
  - n = 27? (after 111 steps)



The Hailstone Turing Machine

- Let Σ = {a} and consider the language L = {a^n | n > 0 and the hailstone sequence terminates for n }

Could we build a TM for L?

- We can build a TM that works as follows:
  - If the input is ε, reject
  - While the string is not **a**:
    - 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 one **a**
  - Accept

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 before will always stop running!
- The conjecture (unproven claim) that the hailstone sequence always terminates is called the ***Collatz Conjecture***
- This problem has eluded a solution for a long time
- Paul Erdős said "Mathematics may not be ready for such problems."

Paul Erdős (1913 – 1996) was a renowned Hungarian mathematician. He was one of the most prolific mathematicians and producers of mathematical conjectures of the 20th century.

Terence Tao FAA FRS (1975) is an Australian-American mathematician. He received the Fields Medal (2006) and Breakthrough Prize at Math. (2014). In 2019, Tao proved that almost all Collatz orbits have finite stopping time.

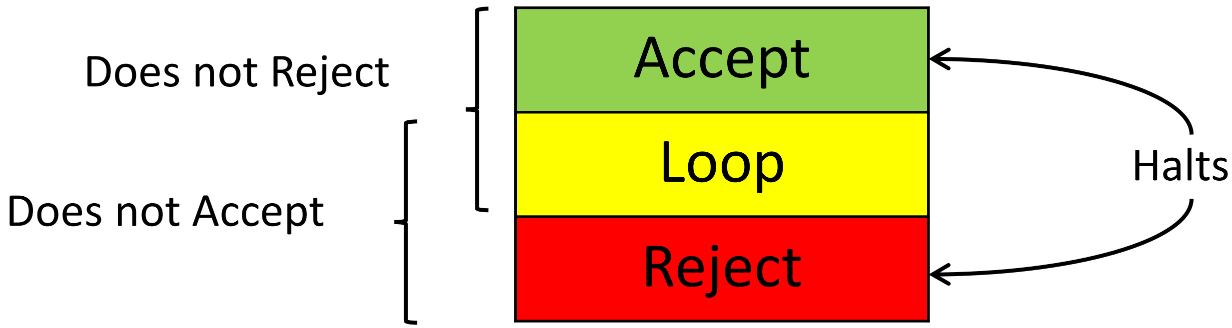
An Important Observation

- Unlike finite automata, which automatically halt after all the input is read, TMs keep running until they explicitly return true or return false
- 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?

Recognizers and Recognizability

Terminology

- Let M be a Turing machine
- M ***accepts*** a string w if it returns true on w
- M ***rejects*** a string w if it returns false on w
- M ***loops infinitely*** (or ***loops***) on a string w if when run on w it neither returns true nor returns false
- M ***does not accept w*** if it either rejects w or loops 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



Recognizers and Recognizability

- A TM M is called a recognizer for a language L over Σ if the following statement is true:

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

- If you are absolutely certain that $w \in L$, then running a recognizer for L on w will (eventually) confirm this
  - Eventually, M will accept w
- If you don't know whether $w \in L$, running M on w may never tell you anything
  - M might loop on w
  - but you can't differentiate between "it'll never give an answer" and "just wait a bit more."
- Does that feel like "solving a problem" to you?

Recognizers and Recognizability

- The hailstone TM M we saw earlier is a recognizer for the language

  L = { a^n | n > 0 and the hailstone sequence terminates for n }
- If the sequence does terminate starting at n, then M accepts a^n
- If the sequence doesn't terminate, then M loops forever on a^n and never gives an answer
- If you somehow knew the hailstone sequence terminated for n, this machine would (eventually) confirm this
- If you didn't know, this machine might not tell you anything.

Recognizer: Examples

```
bool negatives (string input) {
    return false;
}
```

```
bool positives (string input) {
    return true;
}
```

```
bool infinites (string input) {
    while (true) {
        // do nothing
    }
    return false;
}
```

```
bool repeaters (string input) {
    if (input.size() % 2 != 0) return false;
    for (int i = 0; i < input.size() / 2; i++) {
        if (input[2 * i] != input[2 * i + 1])  {
            return false;
        }
    }
    return true;
}
```

- Each of these code is a recognizer for some language. What language does each recognizer recognize?

Recognizers and Recognizability

- Sums of three cubes
- Are there integers x, y, and z where…

  $x^3 + y^3 + z^3 = 10$? Yes! x = 2, y = 1, z = 1

  $x^3 + y^3 + z^3 = 11$? Yes! x = 3, y = -2, z = -2

  $x^3 + y^3 + z^3 = 12$? Yes! x = 7, y = 10, z = -11

  $x^3 + y^3 + z^3 = 13$?

Recognizers and Recognizability

- Surprising fact: until 2019, no one knew whether there were integers x, y, and z where $x^3 + y^3 + z^3$ = 33
- A heavily optimized computer search found this answer:

  x = 8,866,128,975,287,528

  y = -8,778,405,442,862,239

  z = -2,736,111,468,807,040

- As of November 2021, no one knows whether there are integers x, y, and z where $x^3 + y^3 + z^3$ = 114.

Recognizers and Recognizability

- Consider the language L = { a^n | ∃x ∈ ℤ. ∃y ∈ ℤ. ∃z ∈ ℤ. $x^3 + y^3 + z^3$ = n }
- Here's pseudocode for a recognizer to see whether such a triple exists:

```
for max = 0, 1, 2, …
    for x from -max to +max:
        for y from -max to +max:
            for z from -max to +max:
                if x^3 + y^3 + z^3 = n: return true
```

- If you somehow knew there was a triple x, y, and z where x3 + y3 + z3 = n, running this program will (eventually) convince you of this
- If you weren't sure whether a triple exists, this recognizer might not be useful to you

Recognizers and Recognizability

- The class RE consists of all recognizable languages
- Formally speaking:

  RE = { L | L is a language and there's a recognizer for L }

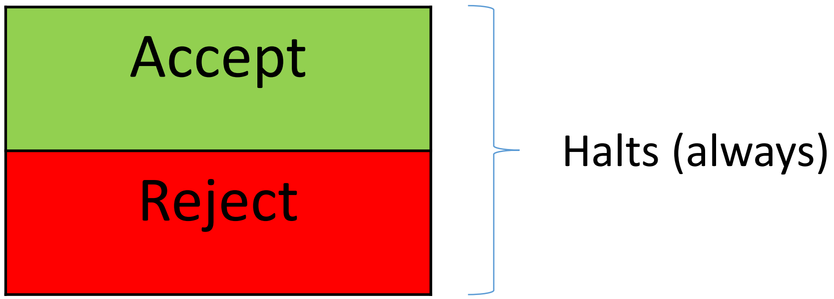
- You can think of RE as "all problems with yes/no answers where "yes" answers can be confirmed by a computer."
  - Given a recognizable language L and a string w $\in L$, running a recognizer for L on w will eventually confirm w $\in L$
  - The recognizer will never have a "false positive" of saying that a string is in L when it isn't
- This is a "weak" notion of solving a problem
- What is a "stronger" one?

Deciders and Decidability

Deciders and Decidability

- Some, but not all, TMs have the following property: the TM halts on all inputs

- If you are given a TM M that always halts, then for the TM M, the statement “M does not accept w” means “M rejects w.”



Deciders and Decidability

- A TM M is called a ***decider*** for a language L over Σ if the following statements are true:

$$\forall w \in \Sigma^*.\ M \text{ halts on } w$$

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

- In other words, M accepts all strings in L and rejects all strings not in L
- In other words, M is a recognizer for L, and M halts on all inputs
- If you aren't sure whether $w \in L$, running M on w will (eventually) give you an answer to that question.

Deciders and Decidability

- The hailstone TM M we saw earlier is a recognizer for the language

  L = { a^n | n > 0 and the hailstone sequence terminates for n }

- If the hailstone sequence terminates for n, then M accepts a^n. If it doesn't, then M does not accept a^n
- We honestly don't know if M is a decider for this language
  - If the hailstone sequence always terminates, then M always halts and is a decider for L
  - If the hailstone sequence doesn't always terminate, then M will loop on some inputs and isn't a decider for L

Recognizer: Examples

```
bool negatives (string input) {
   return false;
}
```

```
bool positives (string input) {
   return true;
}
```

```
bool infinites (string input) {
   while (true) {
       // do nothing
    }
   return false;
}
```

```
bool repeaters (string input) {
   if (input.size() % 2 != 0) return false;
   for (int i = 0; i < input.size() / 2; i++) {
      if (input[2 * i] != input[2 * i + 1])  {
         return false;
      }
   }
   return true;
}
```

- Each of these code is a recognizer for some language. Which are deciders?

Deciders and Decidability

- While no one knows whether there are integers x, y, and z where $x^3 + y^3 + z^3$ = 114, it is very easy to figure out whether there are integers x, y, and z where $x^2 + y^2 + z^2$ = 114

- Why?

Deciders and Decidability

- Consider the language L = $\{a^n \mid \exists x \in \mathbb{Z}. \exists y \in \mathbb{Z}. \exists z \in \mathbb{Z}. x^2 + y^2 + z^2 = n\}$
- Here's pseudocode for a decider to see whether such a triple exists:

```
for x from 0 to n:
    for y from 0 to n:
        for z from 0 to n:
            if x^2 + y^2 + z^2 = n: return true
return false
```

- After trying all possible options, this program will either find a triple that works or report that none exists

Deciders and Decidability

- The class R consists of all decidable languages
- Formally speaking:

  R = { L | L is a language and there's a decider for L }

- You can think of R as "all problems with yes/no answers that can be fully solved by computers."
  - Given a decidable language, run a decider for L and see what happens
  - Think of this as "knowledge creation" – if you don't know whether a string is in L, running the decider will, given enough time, tell you
- The class R contains all the regular languages, all the context-free languages, most of algorithms, etc.
- This is a "strong" notion of solving a problem.

R and RE Languages

- Every decider for *L* is also a recognizer for *L*
- This means that R ⊆ RE
- Hugely important theoretical question:

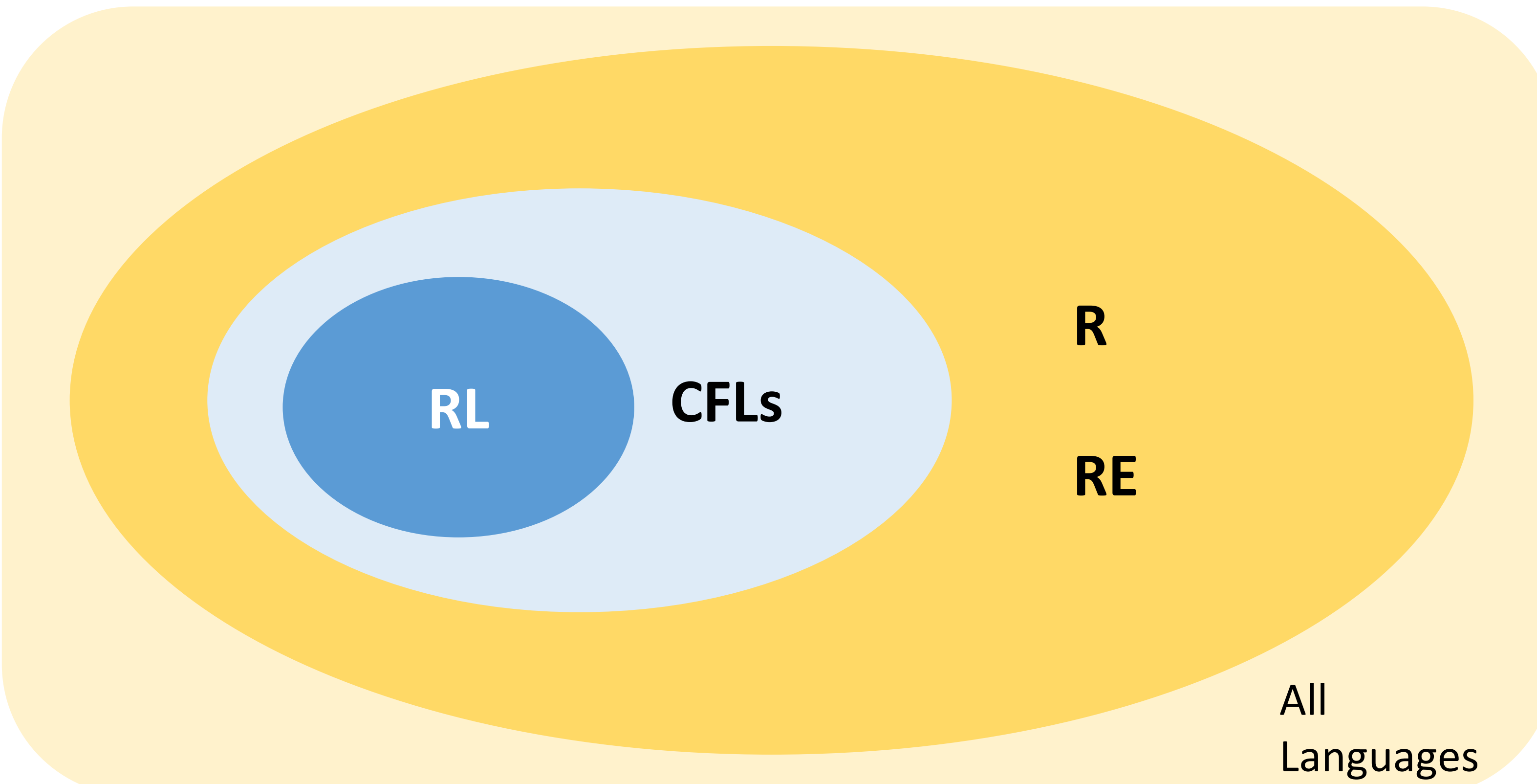
$$\mathrm{R} \stackrel{?}{=} \mathrm{RE}$$

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

Relations

By Definition, R ⊆ RE

R = RE ???



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?