Unsolvable Problems

Part Three

Recap from Last Time

What exactly is the class **RE**?

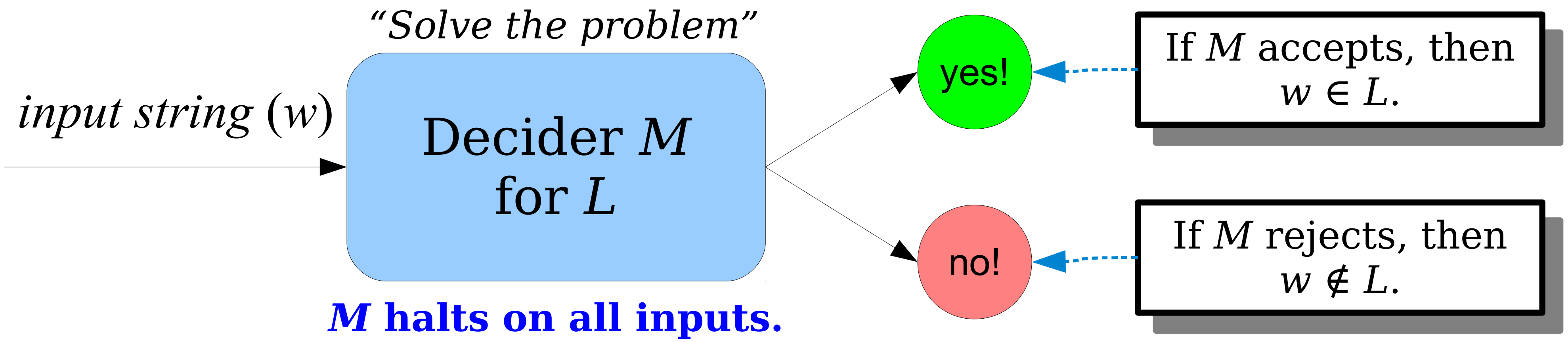
Verifiers

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

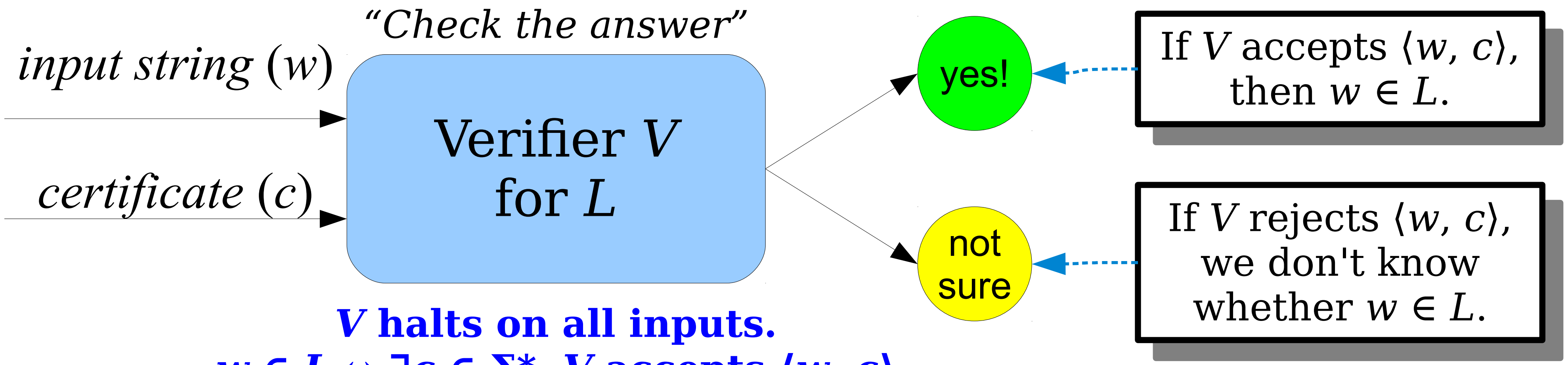
$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*. \ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
  - If V does not accept $\langle w, c \rangle$, then either
    - $w \in L$, but you gave the wrong c, or
    - $w \notin L$, so no possible c will work.

Deciders and Verifiers


"Solve the problem"
input string (w)
Decider M for L
yes!
no!
If M accepts, then $w \in L$.
If M rejects, then $w \notin L$.

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


"Check the answer"
input string (w)
certificate (c)
Verifier V for L
yes!
not sure
If V accepts $\langle w, c\rangle$, then $w \in L$.
If V rejects $\langle w, c\rangle$, we don't know whether $w \in L$.

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

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

RE and Proofs

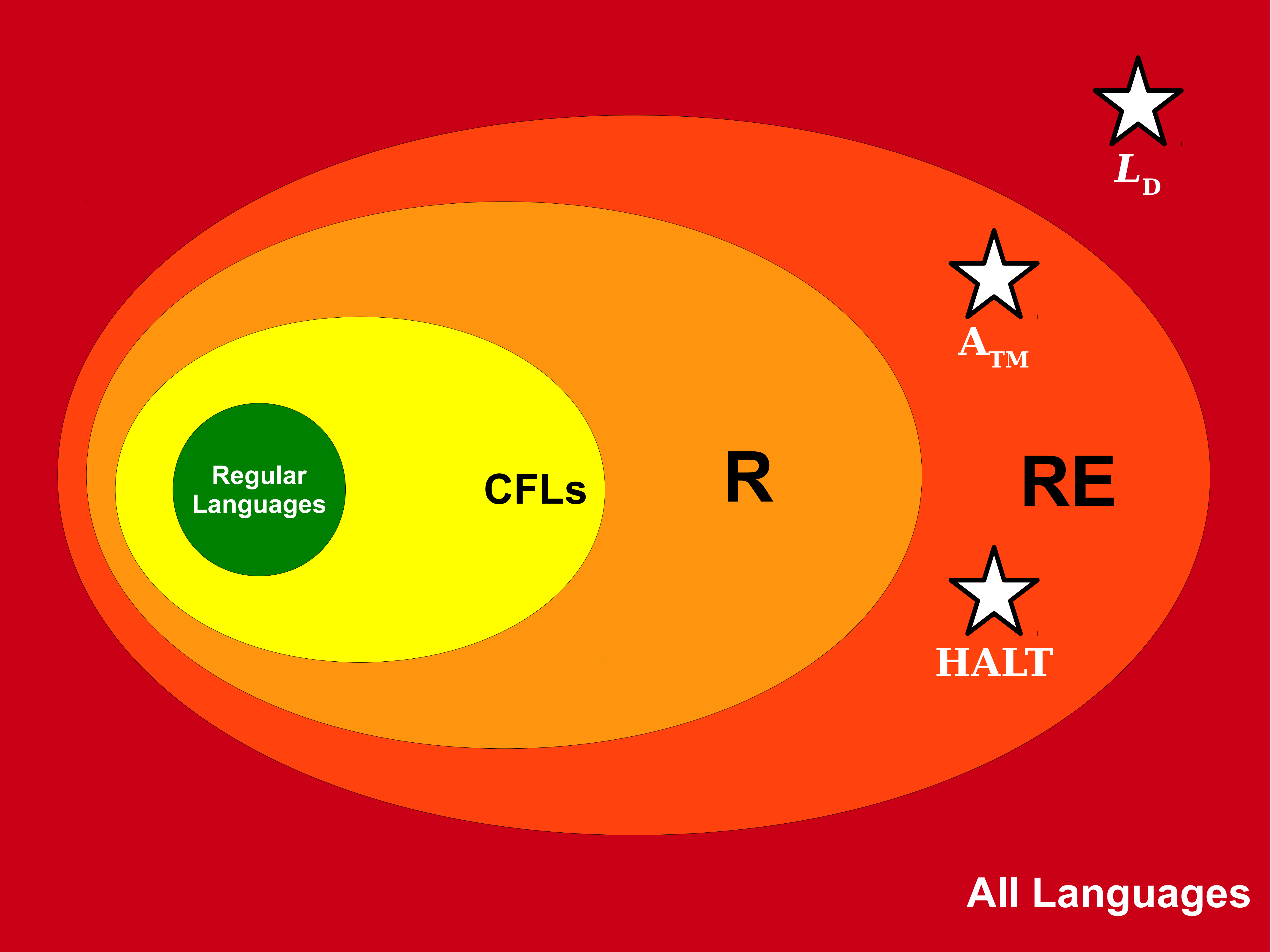
- If the **RE** languages represent languages where membership can be proven, what does a non-**RE** language look like?
- Intuitively, a language is *not* in **RE** if there is no general way to prove that a given string $w \in L$ actually belongs to L.
- In other words, even if you knew that a string was in the language, you may never be able to convince anyone of it!

Diagonalization Revisited

- The ***diagonalization language***, which we denote $\boldsymbol{L_\mathrm{D}}$, is defined as

  $$\boldsymbol{L_\mathrm{D} = \{ \langle M \rangle \mid M \textbf{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}}$$

- That is, L_D is the set of descriptions of Turing machines that do not accept themselves.
- ***Theorem:*** $L_\mathrm{D} \notin \mathbf{RE}$.


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

Finding Non-**RE** Languages

Self-Reference and **RE**

- Following up on our previous tradition, we will use self-reference as our main tool for showing that languages are ***unrecognizable*** (not in **RE**).
- However, things are a bit more complicated when finding non-**RE** languages than finding non-**R** languages.
- Why?
  - **R** is the class of problems that have deciders. If we assume a language is decidable, we assume we have a magic subroutine that always produces the answer we want.
  - **RE** is the class of problems that have verifiers. A verifier for a language can't necessarily tell you whether a string is in a language or not.

The Unhalting Problem

- Consider this problem:

Given a TM *M* and a string *w*,
does *M* loop on *w*?

- As a formal language:

$$\mathit{LOOP} = \{ \langle M, w \rangle \mid M \text{ is a TM that loops on } w \}$$

- How hard of a problem is this to solve?

The Unhalting Problem

- Intuitively, would we expect this language to belong to class **R**?
  - ***No:*** The only general way to figure out what a TM will do is to run it.
  - Can formalize this by building a program that asks if it will loop and does the opposite.
- Intuitively, would we expect this language to belong to class **RE**?
  - ***Remember:*** The only general way to find out what a TM is going to do is to run it.
  - If you ran a TM and you didn't see it accept or reject, that doesn't mean that it's looping! You may just need more time.
  - How would you convince someone that it was looping?

Self-Reference and Verifiers

- Assume for the sake of contradiction that *LOOP* ∈ **RE**.
- This means that there's a verifier for *LOOP*.
- In software, that would be a function like this one:

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate)
```

- This function is our verifier:
  - If `program` loops on `input`, then there is some choice of `certificate` that causes this function to return true.
  - If `program` halts on `input`, this function always returns false, regardless of what `certificate` is.
  - The function always returns a value.

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
   /* … implementation … */
}

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

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if…

… this program loops on its input?

There is a certificate that proves it loops.

So the program halts on its input!

… this program halts on its input?

There is no certificate that proves it loops.

So the program loops on its input!

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

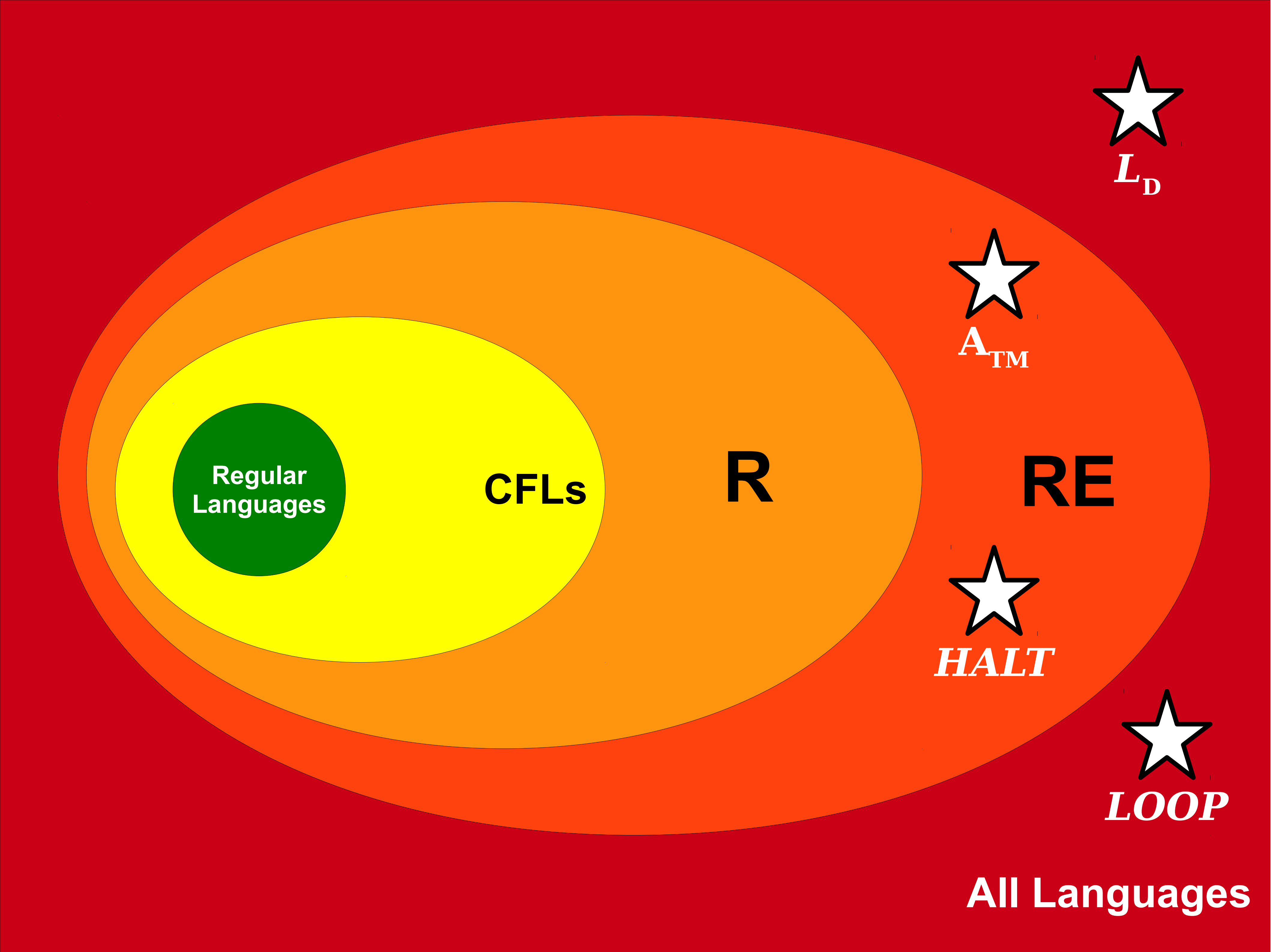
Proof: By contradiction; assume that $LOOP \in \mathbf{RE}$. Then there is some verifier V for $LOOP$. This verifier has the property that if M is a TM that loops on some string w, there is a certificate c such that V accepts $\langle M, w, c\rangle$, and if M halts on w, V will never accept $\langle M, w, c\rangle$ for any certificate c.

Given this, we could then construct the following TM:

```
M = "On input w:
        Have M obtain its own description, ⟨M⟩.
        For all strings c:
            If V accepts ⟨M, w, c⟩, accept.
```

Choose any string w and trace through the execution of the machine. If V ever accepts $\langle M, w, c\rangle$, we are guaranteed that M loops on w, but in this case we find that M accepts w, a contradiction. If V never accepts $\langle M, w, c\rangle$, then we are guaranteed that M halts on w, but in this case we find that M loops infinitely on w, a contradiction.

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



L_D
A_{TM}
Regular Languages
CFLs
R
RE
$HALT$
$LOOP$
All Languages

$LOOP \notin$ **RE**

- The fact that $LOOP \notin$ **RE** gives us a powerful intuition:

There is no general way to prove that a TM will loop on a particular input.

- This is a really useful intuition going forward – it will help you get a better sense for when a problem is likely to be unrecognizable.

Self-Reference and **RE**

- The proof template for showing undecidability via self-reference is the following:
  - Build a machine that asks what it is about to do.
  - Based on the answer, have the machine do the exact opposite.
- The proof template for showing *unrecognizability* via self-reference is the following:
  - Build a machine that tries to *prove* it will do something by sitting in an infinite loop and trying all possible certificates as inputs to a verifier.
  - If it *proves* that it will do something, have it do the opposite.
  - If it can't *prove* that it will, it will loop infinitely. Design the machine so that looping infinitely causes it to behave incorrectly.

Another Non-**RE** Language

- Let's revisit our secure voting problem:

Given a TM *M*, is the language of *M*
{ *w* ∈ {r, d}* | *w* has more r's than d's }?

- We know that this language isn't decidable (we proved that on Wednesday).
- Is it recognizable?
- ***Good intuition:*** If you *knew* that a TM was a secure voting machine, what could you do to prove it to me?

Secure Voting

- Suppose that the secure voting problem is recognizable. Then we could write a function

```
bool imConvincedIsSecure(string program,
                         string certificate)
```

  that would accept as input a program and a certificate, then would check whether that certificate “proves” that the program is a secure voting machine.
- We're going to use this to build a self-referential program that causes problems:
  - If there is a certificate that makes it a secure voting machine, we're going to make it behave in a way that makes it *not* a secure voting machine.
  - If there are no certificates that make it behave like a secure voting machine, we're going to make it behave in a way that makes it a secure voting machine.

```
bool imConvincedIsSecure(string program, string cert) {
   /* … some implementation … */
}

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

   if (countRs(input) > countDs(input)) {
      accept();
   }

    for (i = 0 to infinity) {
      for (each string c of length i) {
         if (imConvincedIsSecure(me, c)) {
            accept();
         }
      }
   }
}
```

What happens if...

... this program is a secure voting machine?

There is a certificate that proves it's secure.

So the program isn't a secure voting machine!

... this program is not a secure voting machine?

There is no certificate that proves it's secure.

So the program is a secure voting machine!

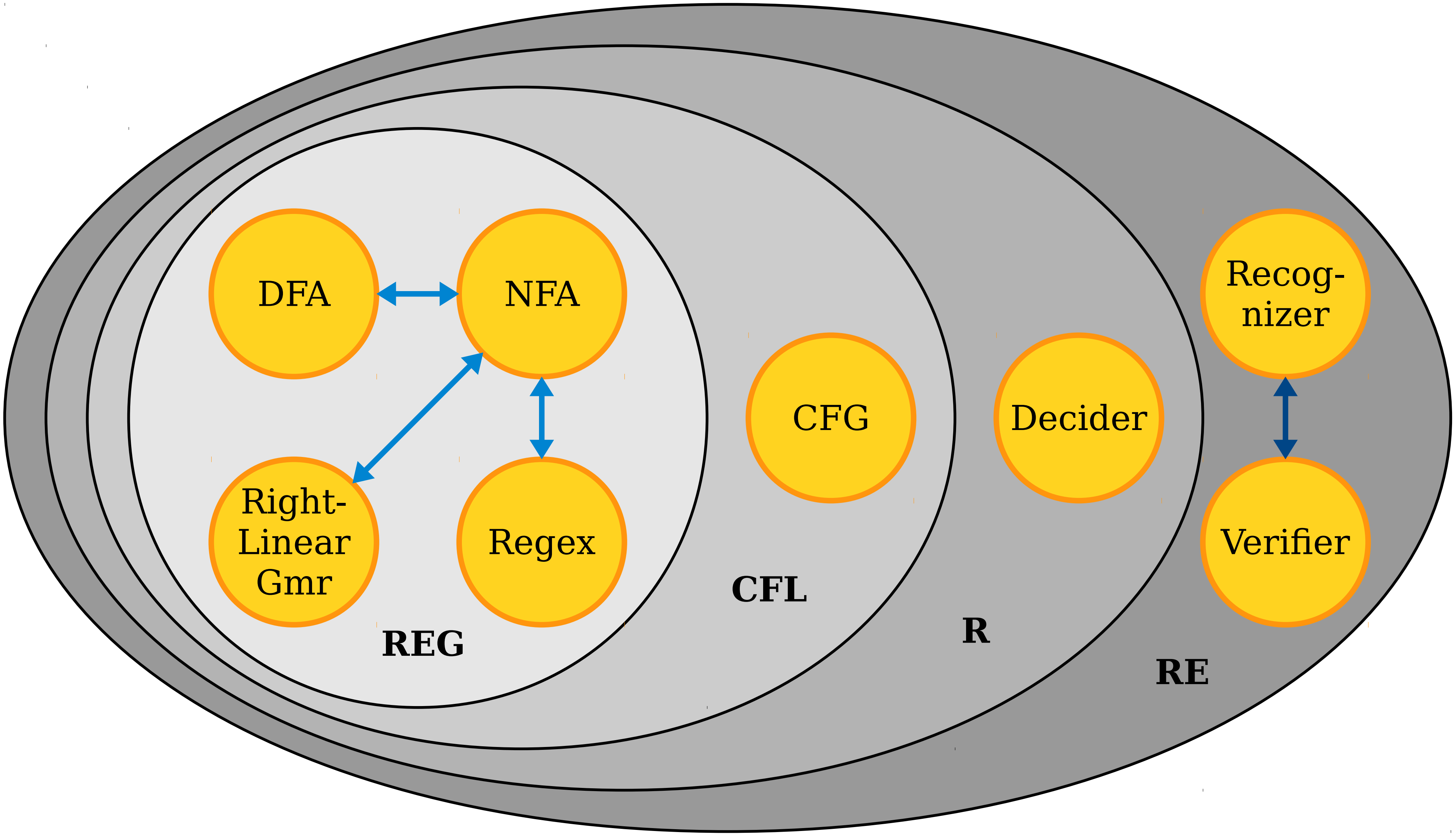
Secure Voting

- The argument we've just made shows that, in general, there's no way to prove that a voting system is secure!
- What options do we have?
  - Restrict the sorts of programs we can write so that they're not as powerful as Turing machines.
  - Have a trusted person write the voting software.
  - Don't allow the program to be so complicated that it's hard to reason about.
  - Rely on the opinions of experts to validate the software.
  - Not trust a computer with elections.
- ***Just because this problem is impossible doesn't mean that we shouldn't try to solve it.*** We just need to recalibrate our expectations.

Where We Stand

- We've just done a crazy, whirlwind tour of computability theory:
  - ***The Church-Turing thesis*** tells us that TMs give us a mechanism for studying computation in the abstract.
  - ***Universal computers*** – computers as we know them – are not just a stroke of luck. The existence of the universal TM ensures that such computers must exist.
  - ***Self-reference*** is an inherent consequence of computational power.
  - ***Undecidable problems*** exist partially as a consequence of the above and indicate that there are statements whose truth can't be determined by computational processes.
  - ***Unrecognizable problems*** also exist partially through self-reference and indicate that there are limits to mathematical proof.

The Big Picture



Where We've Been

- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.

Where We're Going

- The class **P** represents problems that can be solved *efficiently* by a computer.
- The class **NP** represents problems where “yes” answers can be verified *efficiently* by a computer.

Time-Out for Announcements!

Problem Set Nine

- Problem Set Eight was due today at 3:00PM.
  - Last chance to use your late days! You can submit by 3:00PM Monday using late days.
  - There's no extra credit for saving late days – other than being able to get a jump on PS9. ☺
- Problem Set Nine goes out today. It's due on the very last day of class.
  - Play around with verifiers, the limits of the **RE** languages, and the **P** versus **NP** question!
  - Some topics require content from Monday's lecture; they're clearly marked.
  - ***No late submissions accepted***. This is a university policy. Sorry!

Final Exam Practice

- We've posted three sets of cumulative Extra Practice Problems (EPP8 – EPP10) on the course website.
  - Solutions will go out on Monday.
- We've also released a final set of challenge problems (Challenge Problems 3).
- Next week, we'll release two practice final exams as well.
- Need more practice? Just let us know.

Your Questions

"I didn't expect to be interested in CS theory at all (and am still worried that I'm more afraid of programming than I am disinterested in it), but I now want to explore the field more. What are good classes to take for people interested in theory?"

I'm going to talk a lot about this on the last day of class. As a sneak preview, look at CS154 and CS161 as good next steps, and to CS255, CS143, Phil 151, and Math 108 as good places to go if you're interested in taking the material in cool directions.

“Why do you only wear khakis?”

I like khakis! They're comfy and easy to wear.

“what problem do you wish more smart people were working on?”

We desperately need more people with both a CS and legal background to help create sensible legislation for online privacy and computer security. This isn't an "end world hunger"-type problem, but it's something that (1) has huge societal ramifications and (2) is something that I think we could fix with more people working on it.

Back to CS103!

Complexity Theory

It may be that since one is customarily concerned with existence, […] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

It may be that since one is customarily concerned with existence, [...] decidability, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-decidable* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

A Decidable Problem

- ***Presburger arithmetic*** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \to x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \land \forall y.\ (P(y) \to P(y + 1))) \to \forall x.\ P(x)$
- Given a statement, it is decidable whether that statement can be proven from the laws of Presburger arithmetic.
- Any Turing machine that decides whether a statement in Presburger arithmetic is true or false has to move the tape head at least $2^{2^{cn}}$ times on some inputs of length n (for some fixed constant c).

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$

$$2^{2^1} = 4$$

$$2^{2^2} = 16$$

$$2^{2^3} = 256$$

$$2^{2^4} = 65536$$

$$2^{2^5} = 18446744073709551616$$

$$2^{2^6} = 340282366920938463463374607431768211456$$

The Limits of Decidability

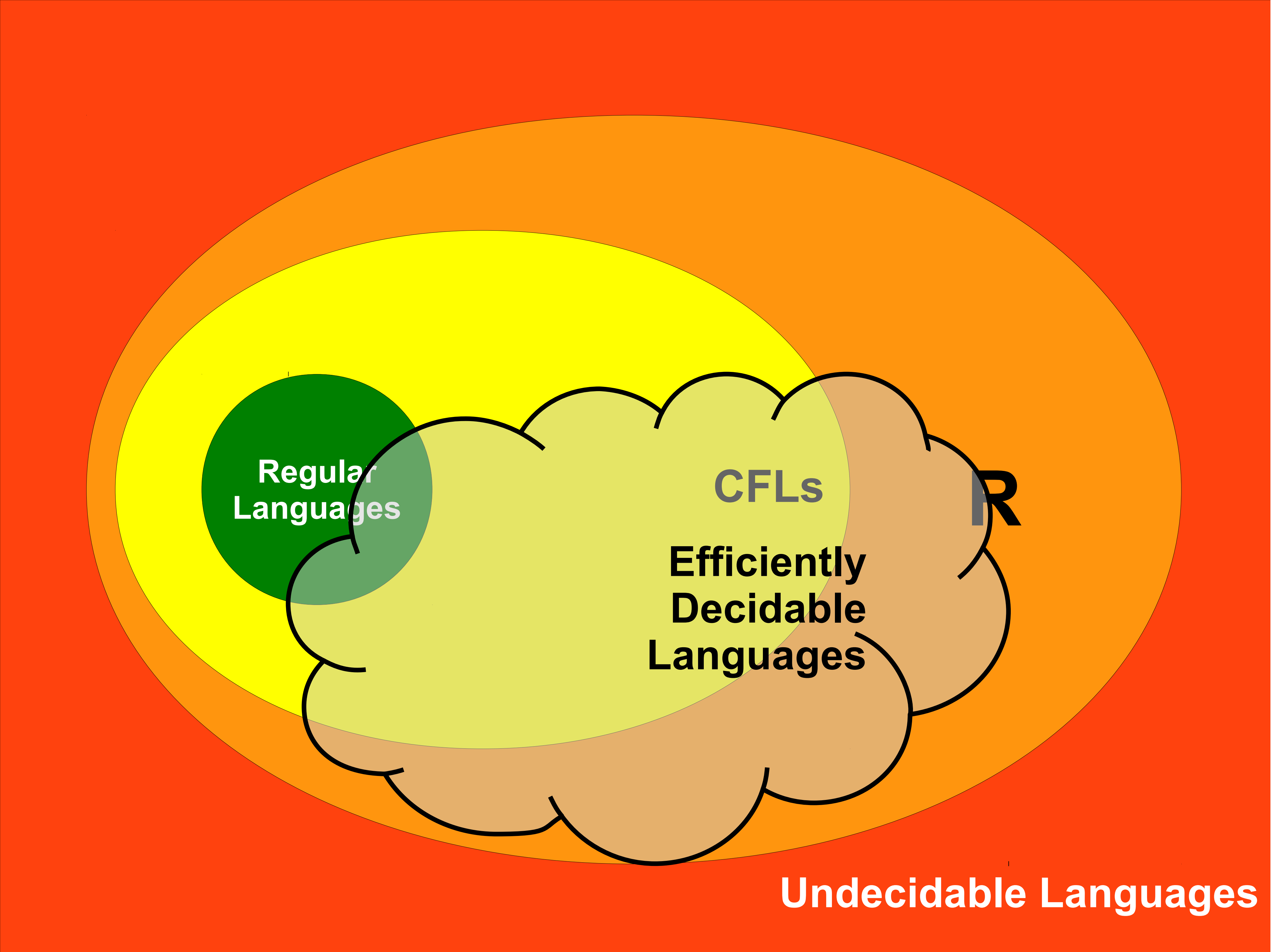
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In ***computability theory***, we ask the question

  What problems can be solved by a computer?

- In ***complexity theory***, we ask the question

  What problems can be solved ***efficiently*** by a computer?

- In the remainder of this course, we will explore this question in more detail.


Regular Languages
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

The Setup

- In order to study computability, we needed to answer these questions:
  - What is “computation?”
  - What is a “problem?”
  - What does it mean to “solve” a problem?
- To study complexity, we need to answer these questions:
  - What does “complexity” even mean?
  - What is an “efficient” solution to a problem?

Measuring Complexity

- Suppose that we have a decider D for some language L.
- How might we measure the complexity of D?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Amount of time required.
  - Number of times a given state is entered.
  - Number of times a given symbol is printed.
  - Number of times a given transition is taken.
  - (Plus a whole lot more...)

What is an efficient algorithm?

Searching Finite Spaces

- Many decidable problems can be solved by searching over a large but finite space of possible options.
- Searching this space might take a staggeringly long time, but only finite time.
- From a decidability perspective, this is totally fine.
- From a complexity perspective, this is totally unacceptable.

A Sample Problem

4	3	11	9	7	13	5	6	1	12	2	8	0	10

Goal: Find the length of the longest increasing subsequence of this sequence.

A Sample Problem

4	3	11	9	7	13	5	6	1	12	2	8	0	10

Goal: Find the length of the longest increasing subsequence of this sequence.

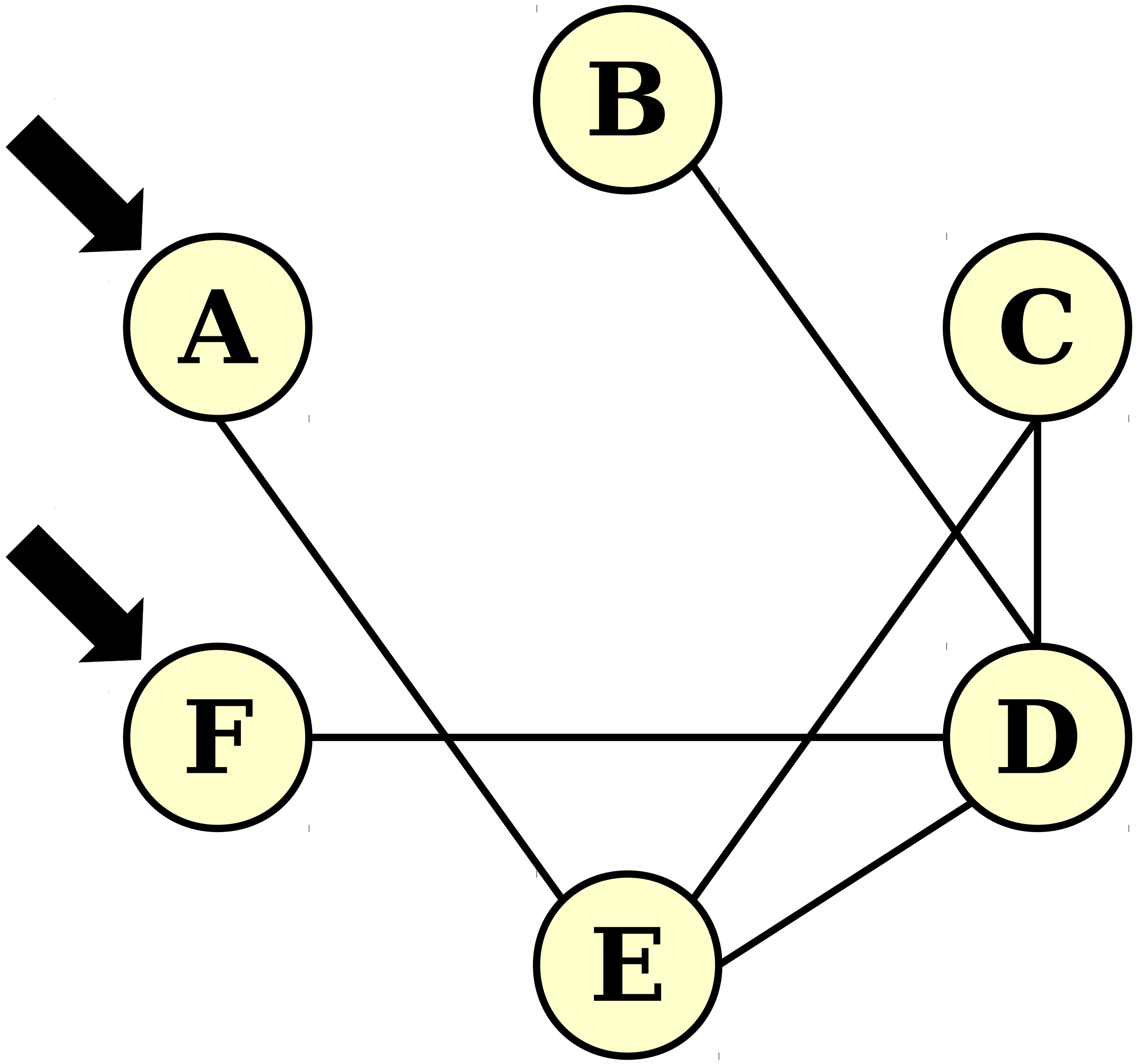
Longest Increasing Subsequences

- ***One possible algorithm:*** try all subsequences, find the longest one that's increasing, and return that.
- There are 2^n subsequences of an array of length n.
  - (Each subset of the elements gives back a subsequence.)
- Checking all of them to find the longest increasing subsequence will take time $O(n \cdot 2^n)$.
- Nifty fact: the age of the universe is about 4.3×10^{26} nanoseconds old. That's about 2^{85} nanoseconds.
- Practically speaking, this algorithm doesn't terminate if you give it an input of size 100 or more.

Longest Increasing Subsequences

- ***Theorem:*** There is an algorithm that can find the longest increasing subsequence of an array in time O(n log n).
- The algorithm is *beautiful* and surprisingly elegant. Look up ***patience sorting*** if you're curious.
- This algorithm works by exploiting particular aspects of how longest increasing subsequences are constructed. It's not immediately obvious that it works correctly.

Another Problem



Goal: Determine the length of the shortest path from **A** to **F** in this graph.

Shortest Paths

- It is possible to find the shortest path in a graph by listing off all sequences of nodes in the graph in ascending order of length and finding the first that's a path.
- This takes time $O(n \cdot n!)$ in an n-node graph.
- For reference: 29! nanoseconds is longer than the lifetime of the universe.

Shortest Paths

- ***Theorem:*** It's possible to find the shortest path between two nodes in an n-node, m-edge graph in time $O(m + n)$.
- This is the breadth-first search algorithm. Take CS106B/X or CS161 for more details!
- The algorithm is a bit nuanced. It uses some specific properties of shortest paths and the proof of correctness is actually nontrivial.

For Comparison

- **Longest increasing subsequence:**
  - Naive: $O(n \cdot 2^n)$
  - Fast: $O(n^2)$
- **Shortest path problem:**
  - Naive: $O(n \cdot n!)$
  - Fast: $O(n + m)$.

Defining Efficiency

- When dealing with problems that search for the “best” object of some sort, there are often at least exponentially many possible options.
- Brute-force solutions tend to take at least exponential time to complete.
- Clever algorithms often run in time O(n), or O(n^2), or O(n^3), etc.

Polynomials and Exponentials

- An algorithm runs in ***polynomial time*** if its runtime is some polynomial in n.
  - That is, time $O(n^k)$ for some constant k.
- Polynomial functions “scale well.”
  - Small changes to the size of the input do not typically induce enormous changes to the overall runtime.
- Exponential functions scale terribly.
  - Small changes to the size of the input induce huge changes in the overall runtime.

The Cobham-Edmonds Thesis

A language L can be ***decided efficiently*** if there is a TM that decides it in polynomial time.

Equivalently, L can be decided efficiently if it can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Like the Church-Turing thesis, this is ***not*** a theorem!

It's an assumption about the nature of efficient computation, and it is somewhat controversial.

The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 - 2n^2 + 4n$
  - $n \log \log n$
- "Efficient" runtimes:
  - $n^{1,000,000,000,000}$
  - 10^{500}
- Inefficient runtimes:
  - 2^n
  - $n!$
  - n^n
- "Inefficient" runtimes:
  - $n^{0.0001 \log n}$
  - 1.000000001^n

Why Polynomials?

- Polynomial time *somewhat* captures efficient computation, but has a few edge cases.
- However, polynomials have very nice mathematical properties:
  - The sum of two polynomials is a polynomial. (Running one efficient algorithm after the other gives an efficient algorithm.)
  - The product of two polynomials is a polynomial. (Running one efficient algorithm a “reasonable” number of times gives an efficient algorithm.)
  - The *composition* of two polynomials is a polynomial. (Using the output of one efficient algorithm as the input to another efficient algorithm gives an efficient algorithm.)

The Complexity Class **P**

- The ***complexity class*** **P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

  P = { *L* | There is a polynomial-time decider for *L* }

- Assuming the Cobham-Edmonds thesis, a language is in **P** if it can be decided efficiently.