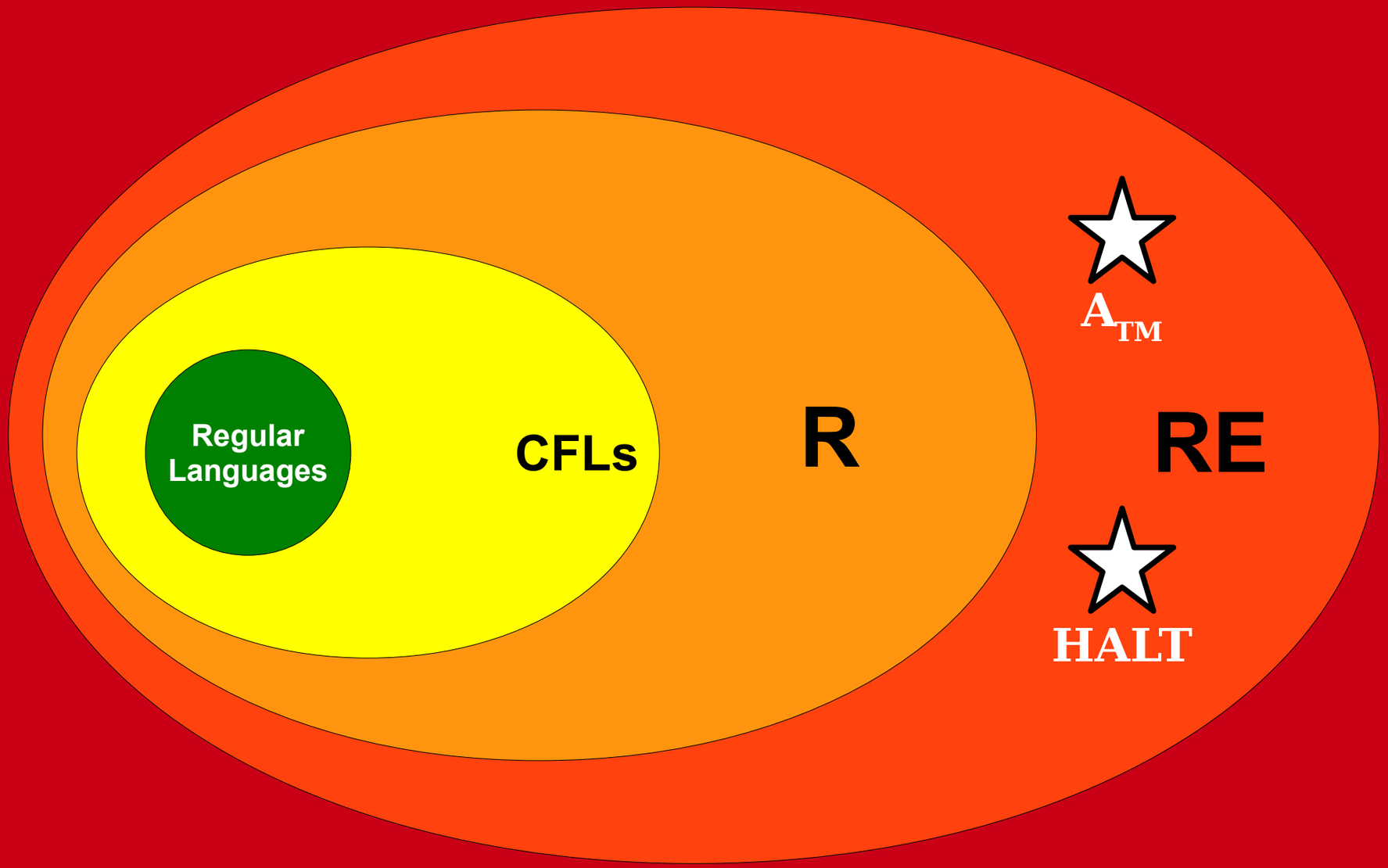
# The Lava Diagram (Wrap-Up of Undecidability)

# RE

(Lava Diagram Problem Tips)




***How do you identify an RE language?***

# RE
## (Lava Diagram Problem Tips)

- You've seen two examples of RE (but not R) problems so far: $A_{TM}$ and *HALT.* They both have this form:
  - $L = \{\langle M,w\rangle \mid$ /* something about $M$'s behavior on $w$ */$\}$
- You've also seen how we used the Trickster approach to prove both are undecidable (not in R).
- *You may have found yourself thinking, "Couldn't I always just use Trickster on <u>any</u> language that looks like* $\langle M\rangle$ *or* $\langle M,w\rangle$ *or similar, to show it's undecidable?"*
- *And the answer, in fact, is* ***yes!*** *(with a couple caveats)*

# Rice's Theorem

(Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- So languages matching these templates will always be **undecidable**:
  - $L = \{\langle M,w\rangle \mid$ /* something about $M$'s behavior on $w$ */$\}$
  - $L = \{\langle M\rangle \mid$ /* something about $M$'s behavior */$\}$
  - $L = \{\langle M\rangle \mid$ /* something about $M$'s language */$\}$
  - $L = \{\langle M_1, M_2\rangle \mid$ /* something comparing $M_1$ and $M_2$'s languages or behaviors */$\}$

**Reminder:** Undecidable means not in R. At best in RE, maybe not even that.

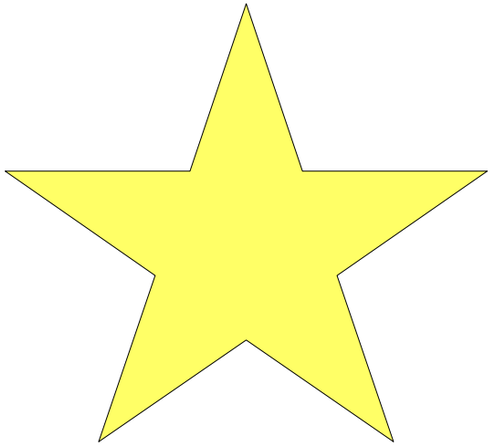
- Example language or behavior criteria: "$M$ accepts at least one string," "$M$'s language is finite," "$M$ loops on $w$"

# Rice's Theorem

(Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 1:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable; only criteria about the TM's language or behavior.
- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually filters. In other words the criteria can't be true of zero TMs or all TMs.



# Rice's Theorem

## (Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 1:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable; only criteria about the TM's language or behavior.
- Questions relating purely to the *text* of the code, and not its *behavior when run* are generally decidable. Decidable examples:
  - $L_1 = \{\langle M,w \rangle \mid$ the string $\langle M \rangle$ contains more a's than the string $w\}$
  - $L_2 = \{\langle M \rangle \mid$ the string $\langle M \rangle$ has odd length$\}$
  - *Intuition: You don't need to try running the TM M to see how many a's its code has or that its code is an odd-length string.*

# Rice's Theorem

## (Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually filters. In other words the criteria can't be true of zero TMs or all TMs.
  - $L = \{\langle M \rangle \mid$ /* criteria true of all TMs */$\} = \Sigma^*$, a regular language!
    - *Example:* $\{\langle M \rangle \mid$ /* $M$ has at least one state */$\}$
  - $L = \{\langle M \rangle \mid$ /* criteria true of zero TMs */$\} = \varnothing$, also a regular language!
  - *Intuition: A problem can't be hard if there's no real question there.*

# Rice's Theorem
## (Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and <u>filters</u> those TMs based on a criteria that has anything to do with the TM's <u>language or behavior</u>, is undecidable.**

- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually <u>filters</u>. In other words the criteria can't be true of zero TMs or all TMs.
  - $L = \{\langle M \rangle \mid$ /* criteria true of all TMs */$\} = \Sigma^*$, a regular language!
    - *Example:* $\{\langle M \rangle \mid$ /* $M$ has at least one state */$\}$
  - $L = \{\langle M \rangle \mid$ /* criteria true of zero TMs */$\} = \emptyset$, also a
    - Example: ??
  - *Intuition: A problem can't be hard if there's no real question there.*

**PollEv:**
Give an example of a criteria that is true of zero TMs.

# Rice's Theorem

## (Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 1:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable; only criteria about the TM's language or behavior.
- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually filters. In other words the criteria can't be true of zero TMs or all TMs.

# Rice's Theorem

(Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 1:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable; only criteria about the TM's language or behavior.
- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually filters. In other words the criteria can't be true of zero TMs or all Tms.

- $L = \Sigma^*$

**PollEv:**
Is L decidable or undecidable?
(Pset9 Q5 part 1)

# Rice's Theorem

(Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 1:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable; only criteria about the TM's language or behavior.
- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually filters. In other words the criteria can't be true of zero TMs or all Tms.

- $L = \{\langle M \rangle \mid M$ is a TM and $M$ is a recognizer for $\emptyset\}$

**PollEv:**
Is L decidable or undecidable?
(Pset9 Q5 part 6)

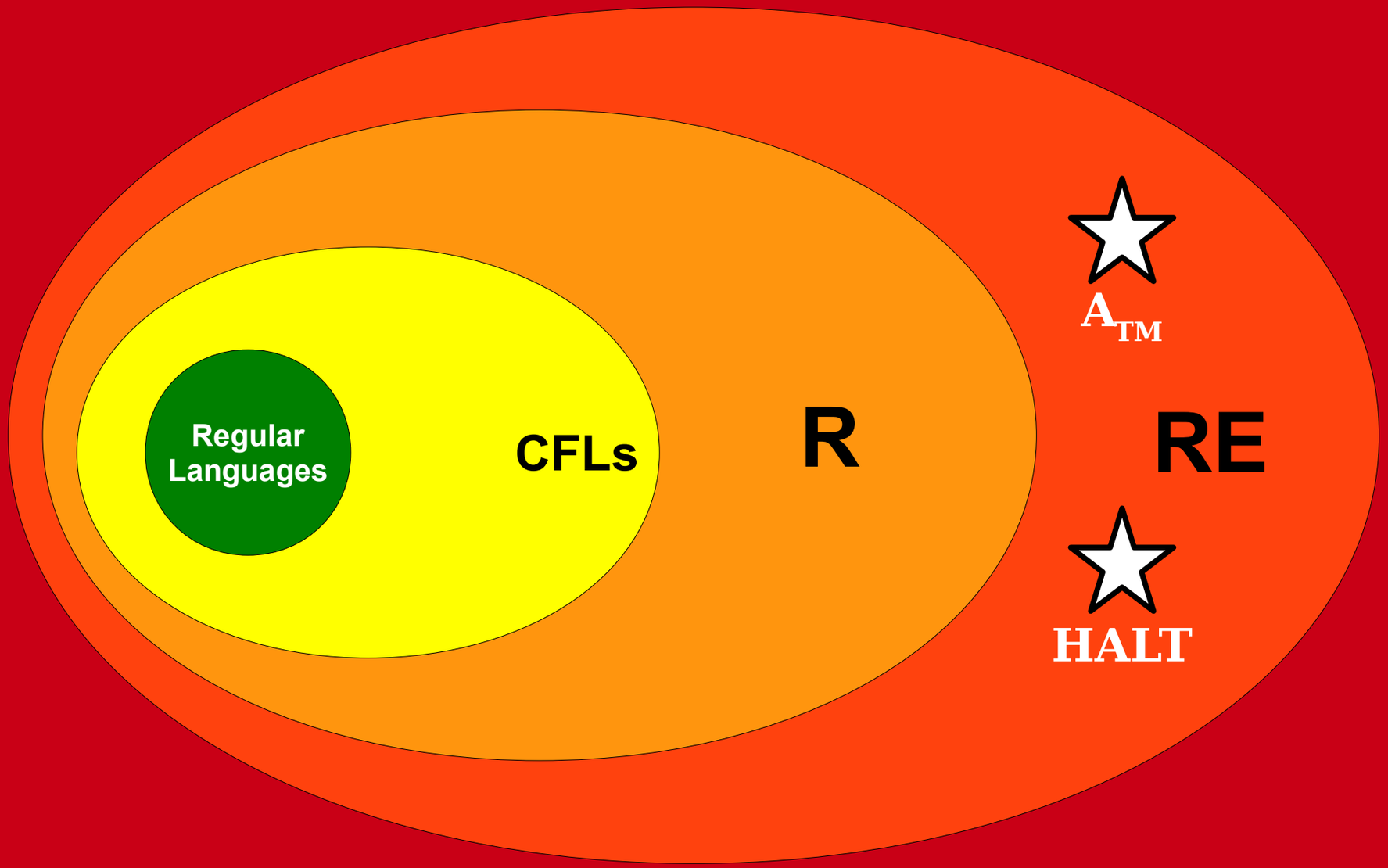
# Rice's Theorem

(Lava Diagram Problem Tips)

**Any language that consists of strings that are TM code, and filters those TMs based on a criteria that has anything to do with the TM's language or behavior, is undecidable.**

- **Caveat 1:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable; only criteria about the TM's language or behavior.
- **Caveat 2:** not all languages of the form $L = \{\langle M \rangle \mid$ /* criteria */$\}$ are undecidable, only if the criteria actually filters. In other words the criteria can't be true of zero TMs or all TMs.

**PollEv:**
Pset9 Q5 parts 9-12: how many are undecidable?




***Rice's Theorem helps us identify what is outside R (may or may not be in RE).***

***What is outside RE?***

# Beyond RE

# How Many Problems Are Outside the Reach of Turing Machines?

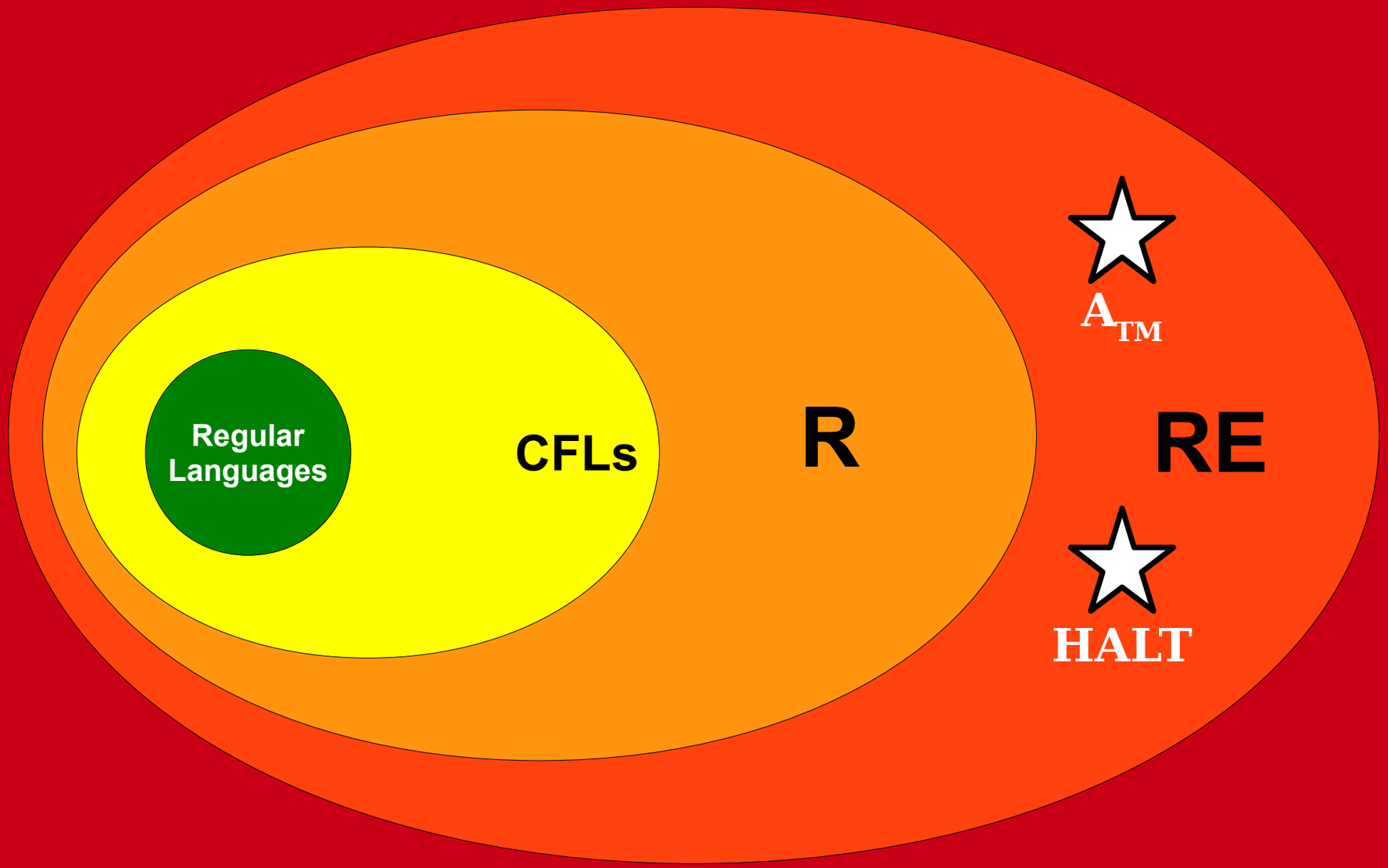
- It’s sad to think about problems that no TM can even *recognize!*
  - Not just undecidable but *unrecognizable!*
- How many are there? Hopefully only a few!

# How Many Problems Are Outside the Reach of Turing Machines?

- A TM is fundamentally just a string (a piece of code). Let's say we can interpret any string as a TM (if not syntactically correct according to some TM code system, we'll just say it is a TM that always rejects).
- So the number of TMs that exist is $|\Sigma^*|$ (the count of all possible strings).
- The number of languages that exist is $|\wp(\Sigma^*)|$ (the count of all possible sets of strings).

# How Many Problems Are Outside the Reach of Turing Machines?

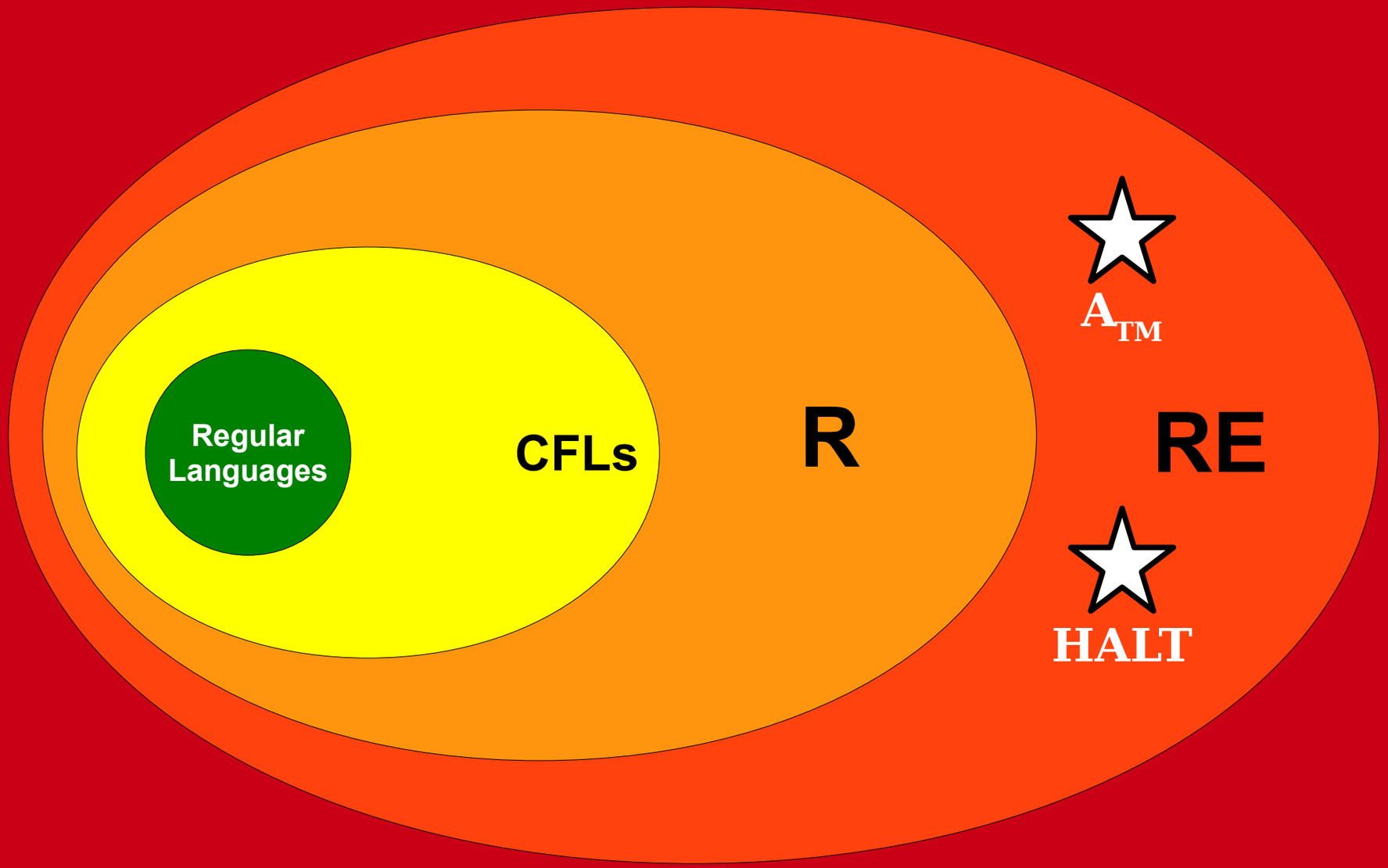
- Sadly, Cantor's Theorem says $|\Sigma^*| < |\wp(\Sigma^*)|$.
- Not nearly enough Turing machines to go around for all the languages that exist.
- So there are many problems (languages) that we can't solve with TMs.
  - *Infinitely many!!*




***What does an outside- RE problem look like?***

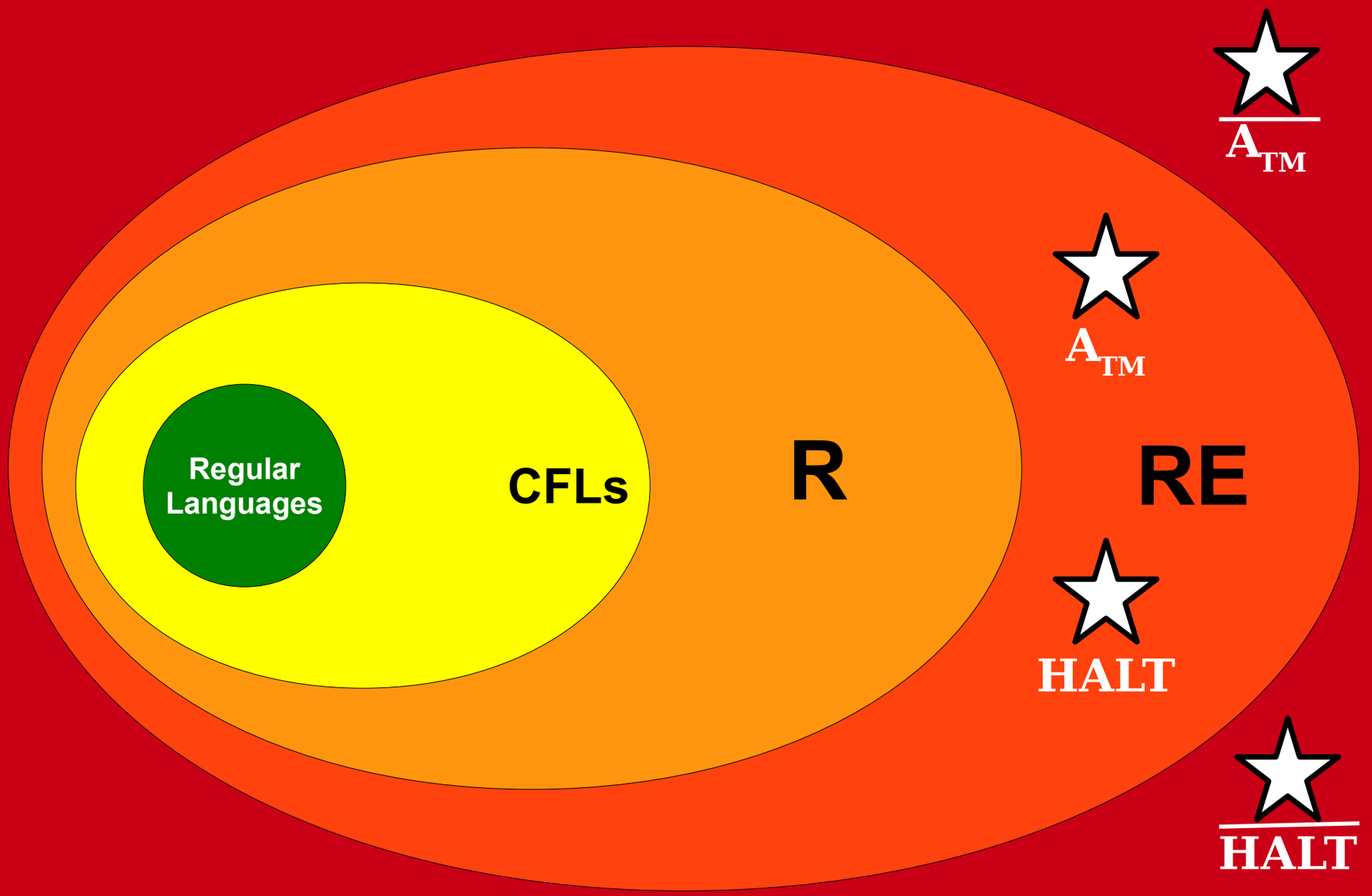
# The Class Unrecognizable
(Lava Diagram Category “All Languages”)

- Languages $L$ that are outside RE are those where:
  - We cannot even build a recognizer TM $M$, where $\mathcal{L}(M) = L$.
  - We cannot build a verifier for $L$.
    - *i.e., there is no finite-length certificate/hint that proves a string is in $L$.*
- Fun fact: this class includes all the complements of undecidable RE languages.




# *What makes a language non-RE?*

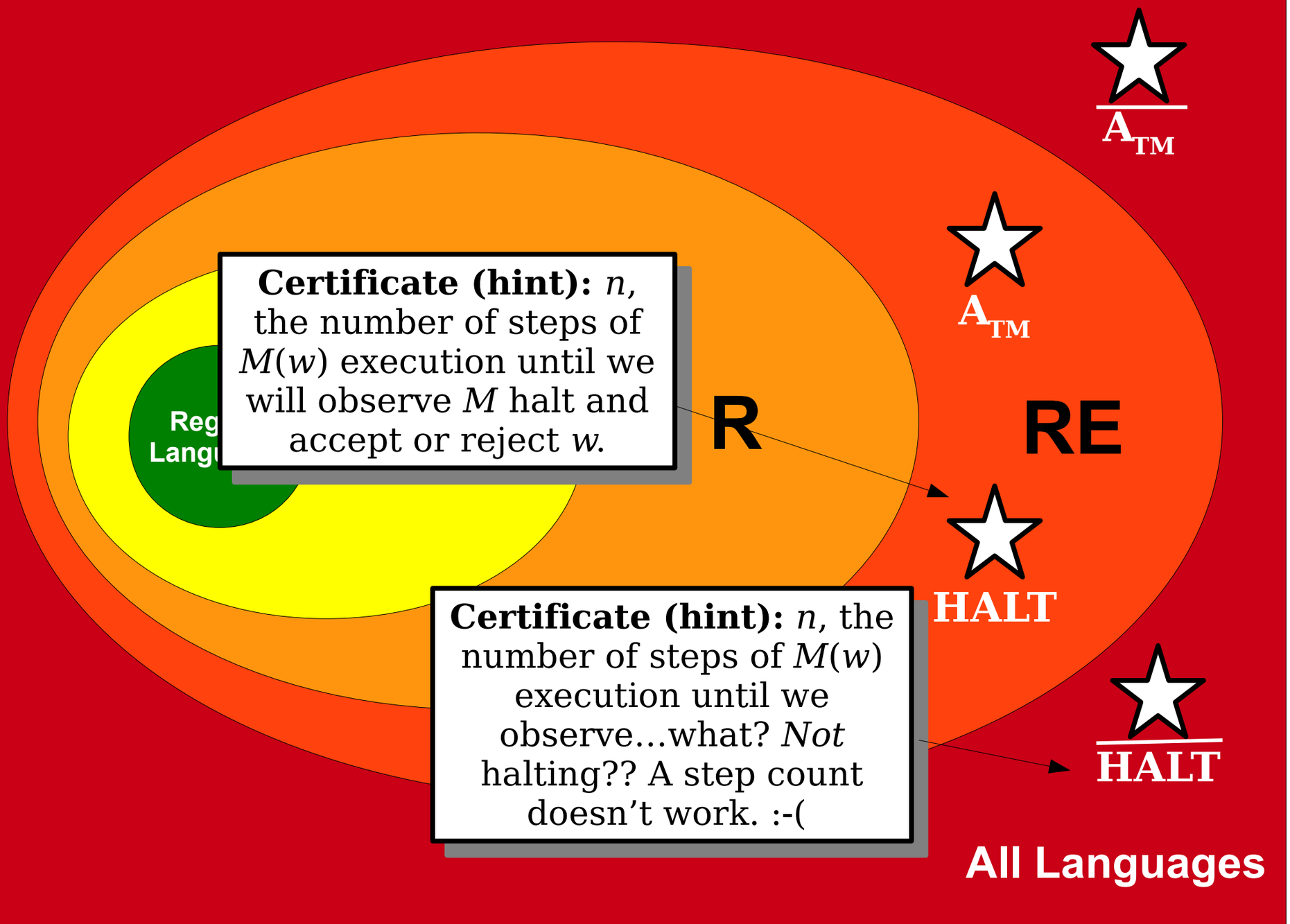
In our discussion of verifiers, we've brushed up against one reason a language could be unverifiable: it's inherently hard to provide a piece of evidence to prove a negative.




# *What makes a language non-RE?*

In our discussion of verifiers, we've brushed up against one reason a language could be unverifiable: it's inherently hard to provide a piece of evidence to prove a negative.

**The complements of undecidable RE languages *are always outside of RE.***




**Certificate (hint):** $n$, the number of steps of $M(w)$ execution until we will observe $M$ halt and accept or reject $w$.

**Certificate (hint):** $n$, the number of steps of $M(w)$ execution until we observe…what? *Not* halting?? A step count doesn't work. :-(

# *What makes a language non-RE?*

In our discussion of verifiers, we've brushed up against one reason a language could be unverifiable: it's inherently hard to provide a piece of evidence to prove a negative.

**The complements of undecidable RE languages *are always outside of RE.***

# The Class Unrecognizable
(Lava Diagram Category “All Languages”)

- Example languages:
  - The complements of undecidable RE languages.
  - $EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid \text{where } \mathcal{L}(M_1) = \mathcal{L}(M_2) \}$
  - $L = \{ \langle M \rangle \mid M \text{ loops on at least five strings} \}$
- For each of the above, ask yourself: what certificate/hint could possibly help?
  - *Of course, not being able to think of one isn't a proof that these are unrecognizable, but it's a good intuition for judging.*

# The Class Unrecognizable
## (Lava Diagram Category “All Languages”)

- Example languages:
  - The complements of undecidable RE languages.
  - $EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid \text{where } \mathcal{L}(M_1) = \mathcal{L}(M_1) \}$
  - $L = \{ \langle M \rangle \mid M \text{ loops on at least five strings} \}$

**PollEv:** A very similar language is: $L' = \{ \langle M \rangle \mid M$ **accepts** at least five strings $\}$. How would you categorize it on the Lava Diagram?

# The Class Unrecognizable
(Lava Diagram Category “All Languages”)

- Example languages:
  - The complements of undecidable RE languages.
  - $EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid \text{where } \mathcal{L}(M_1) = \mathcal{L}(M_1) \}$
  - $\{ \langle M \rangle \mid M \text{ loops on at least five strings} \}$
    - Note that we *can* write a verifier for the language $\{ \langle M \rangle \mid M$ **accepts** at least five strings $\}$, using the certificate $\langle w_1, n_1, w_2, n_2, w_3, n_3, w_4, n_4, w_5, n_5 \rangle$, which is a mouthful, but still finite.

**PollEv:** A very similar language is: $L' = \{ \langle M \rangle \mid M$ **accepts** at least five strings $\}$. How would you categorize it on the Lava Diagram?

# The Language $L_D$

- $L_D = \{\langle M \rangle \mid M \text{ does not accept } \langle M \rangle \}$

  $= \{\langle M \rangle \mid \langle M \rangle \notin \mathcal{L}(M)\}$

- This is a classic example of an unrecognizable language.
- To see why we can't build a TM for this language, ask yourself:
  - If you did build a TM for $L_D$ called $M_{LD}$ would $M_{LD}$ accept $\langle M_{LD} \rangle$?

# Happy Story Time

In a certain isolated town, every house has a lawn and the city requires them all to be mowed. The town has only one gardener, who is also a resident of the town, and this gardener mows the lawns of residents iff they do not mow their own lawn.

# Happy Story Time

In a certain isolated town, every house has a lawn and the city requires them all to be mowed. The town has only one gardener, who is also a resident of the town, and this gardener mows the lawns of residents iff they do not mow their own lawn.

**True or false:** The gardener mows their own lawn.

# The Language $L_D$

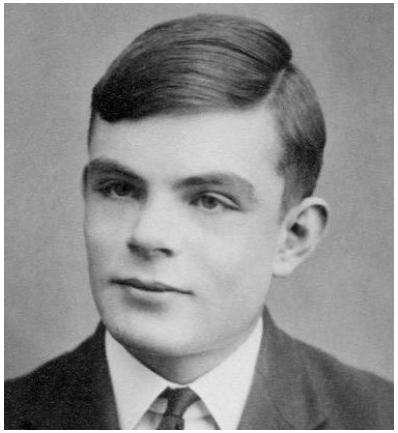
- $L_D = \{\langle M \rangle \mid M \text{ does not accept } \langle M \rangle \}$

  $= \{\langle M \rangle \mid \langle M \rangle \notin \mathcal{L}(M)\}$

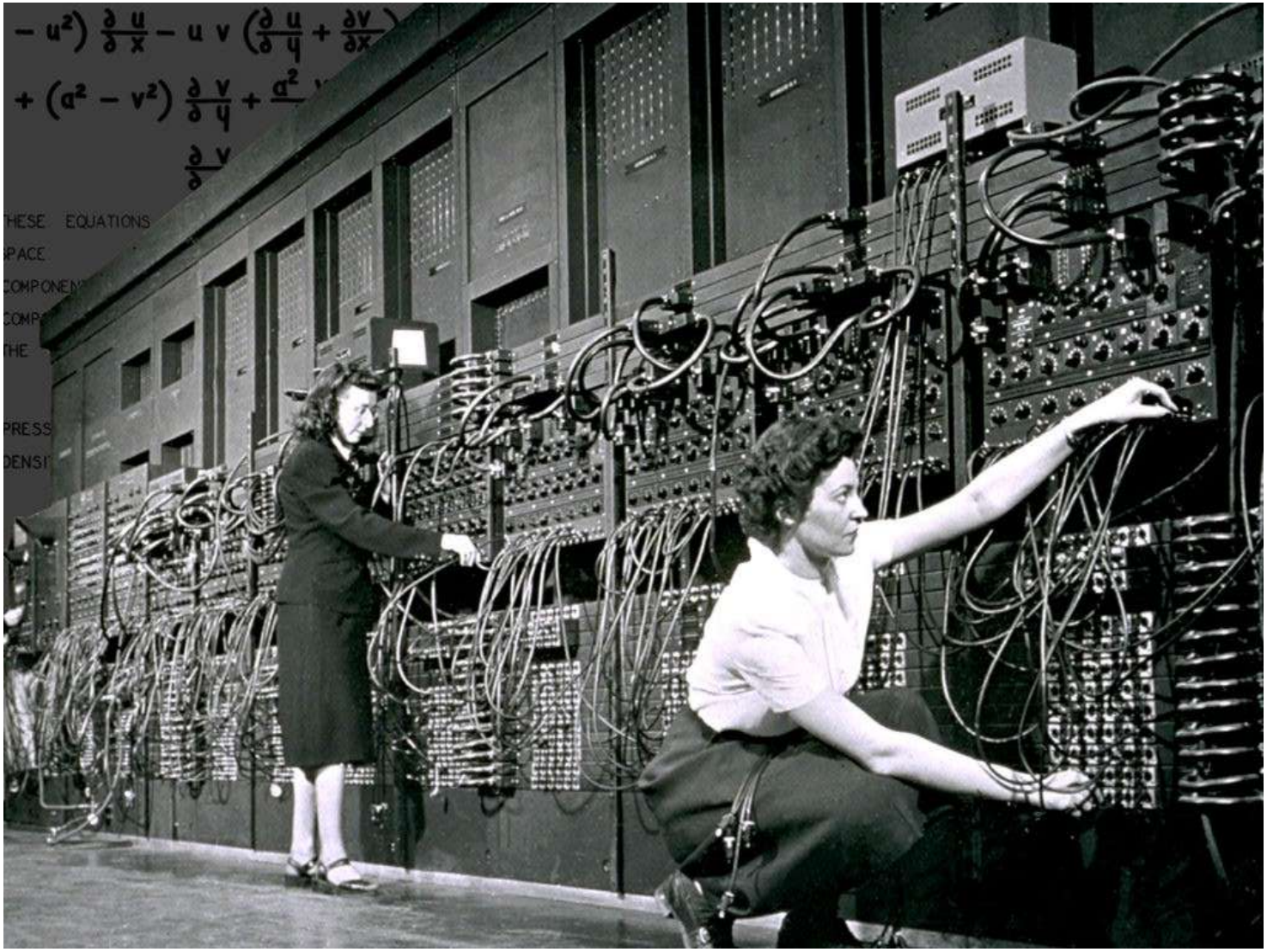
- This is a classic example of an unrecognizable language.
- To see why we can't build a TM for this language, ask yourself:
  - If you did build a TM for $L_D$, called $M_{LD}$, would $M_{LD}$ accept $\langle M_{LD} \rangle$?

The existence of a TM for $L_D$ creates a contradiction, so that's our proof that a TM for $L_D$ cannot exist.

# Complexity Theory



Alan Turing
*On Computable Numbers*
1936




ENIAC
*First General-Purpose Electronic Digital Computer*
1945

# A Decidable Problem

- Consider the following problem:

  ***Given two regular expressions $R_1$ and $R_2$, determine whether $R_1$ and $R_2$ have the same language.***

- This problem is indeed decidable.
  - We autograded your regular expressions in Problem Set Seven. The algorithm we used is 100% accurate.
- ***Theorem:*** There is no algorithm for solving this problem whose runtime is $O(2^{m+n})$, where $m$ and $n$ are the lengths of the input regular expressions.

# 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?

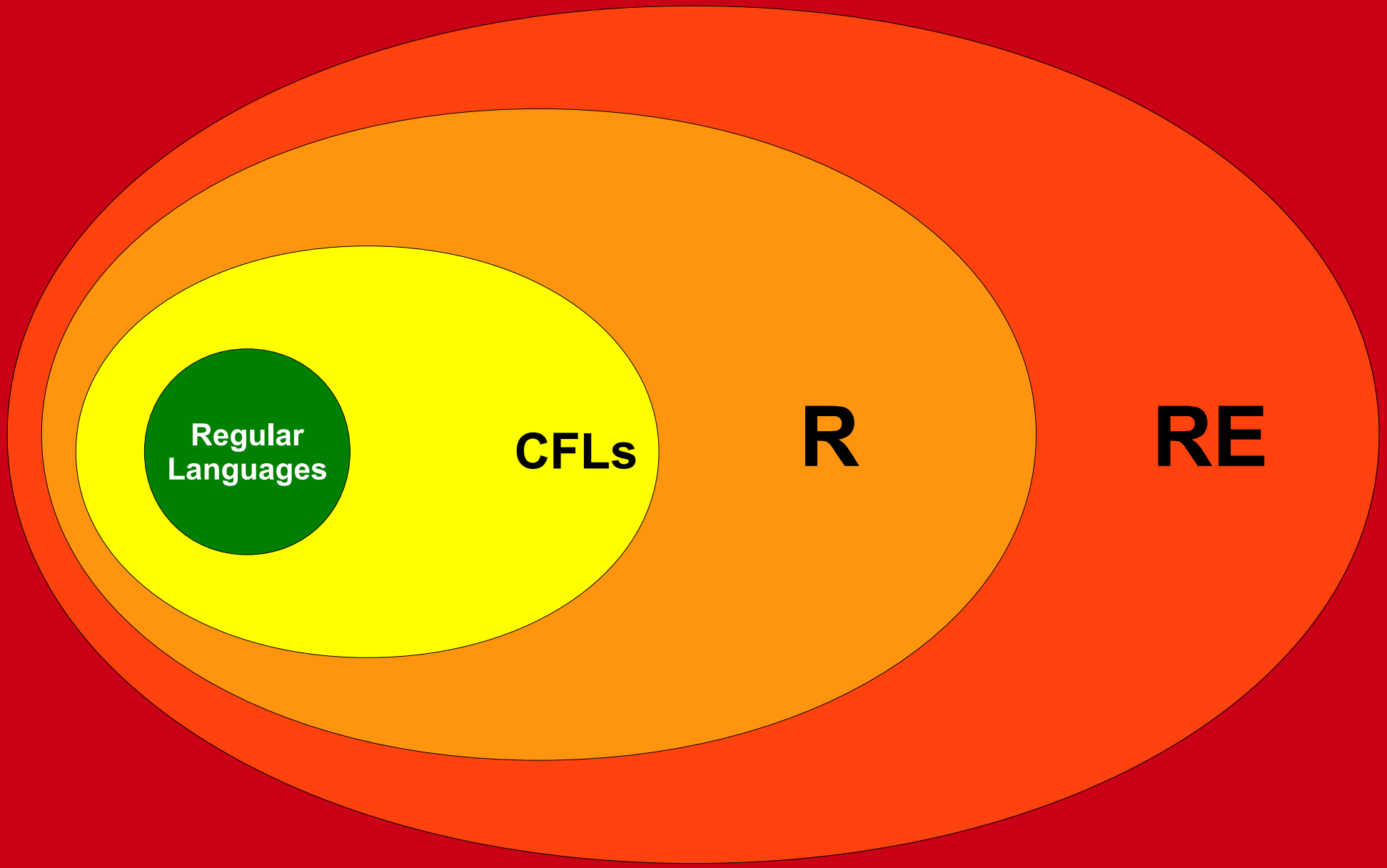
- In the remainder of this course, we will explore this question in more detail.

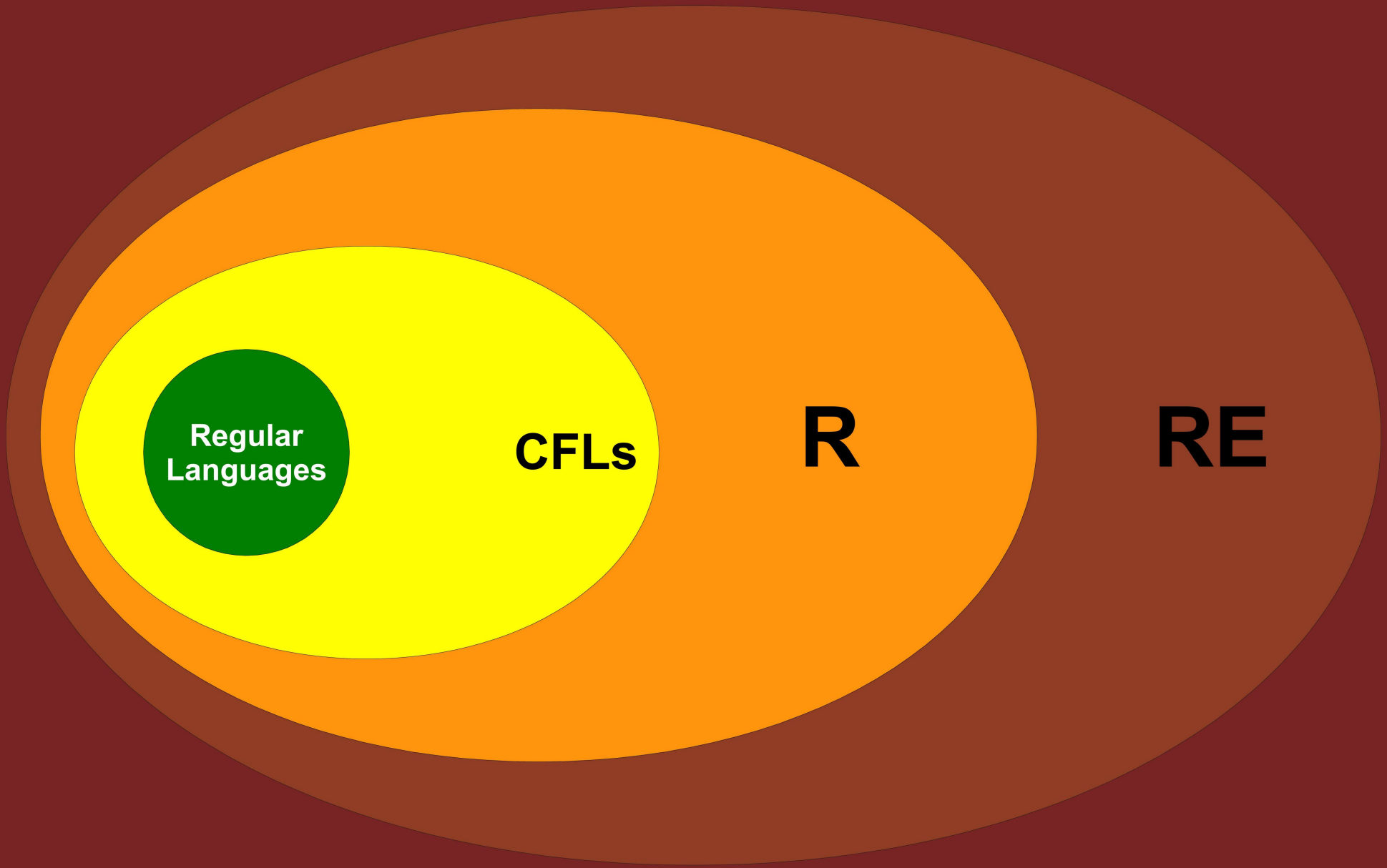
# Where We've Been

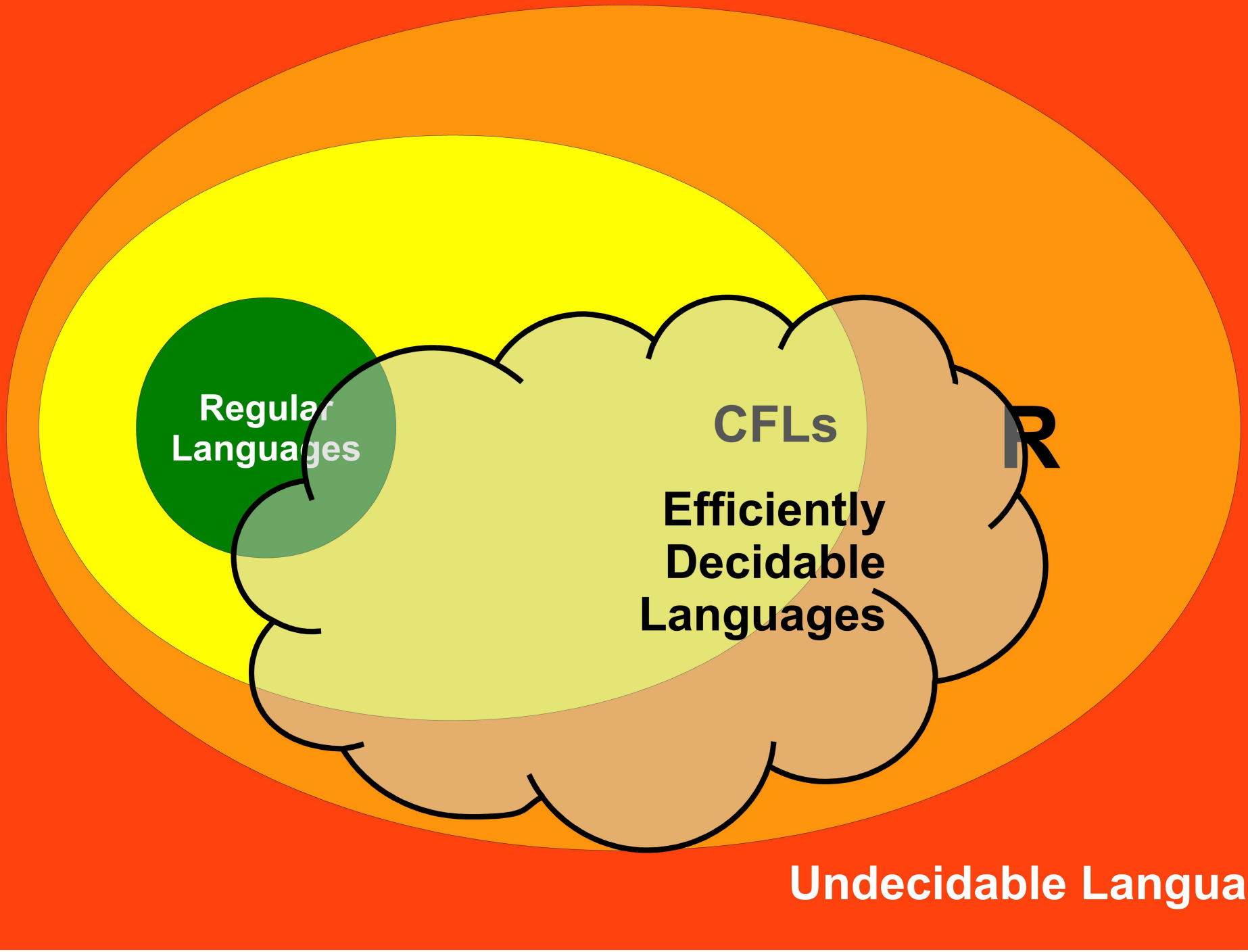
- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.

# 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.


Regular Languages
CFLs
R
RE
All Languages


Regular Languages
CFLs
R
RE
All Languages


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 resources do we want our programs to make “efficient” use of?
  - How do we draw the line between “efficient” and “inefficient?”

# Efficiency

- We have a program written in your Favorite Programming Language that's a decider for some problem.
- What aspect of that program might quantify "efficiency?"
  - The number of lines of code in the program.
  - How deeply-nested the loops or recursion in the program are.
  - How much time it takes for the program to solve the problem.
  - How much memory it takes for the program to solve the problem.
  - How much power it takes for the program to solve the problem.
  - How much network communication it takes for the program to solve the problem.
  - …

# Efficiency

- We have a program written in your Favorite Programming Language that's a decider for some problem.
- What aspect of that program might quantify "efficiency?"
  - The number of lines of code in the program.
  - How deeply-nested the loops or recursion in the program are.
  - How much time it takes for the program to solve the problem.
  - How much memory it takes for the program to solve the problem.
  - How much power it takes for the program to solve the problem.
  - How much network communication it takes for the program to solve the problem.
  - …

What is an efficient algorithm?

Let’s explore some problems and solutions and see what we notice!

# A Common Pattern: 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 may be totally unacceptable.

# A Sample Problem

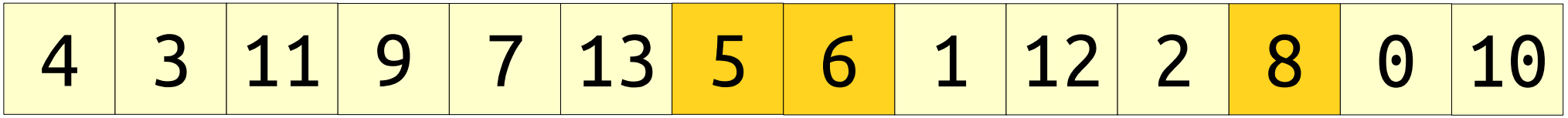
| 4 | 3 | 11 | 9 | 7 | 13 | 5 | 6 | 1 | 12 | 2 | 8 | 0 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# A Sample Problem

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

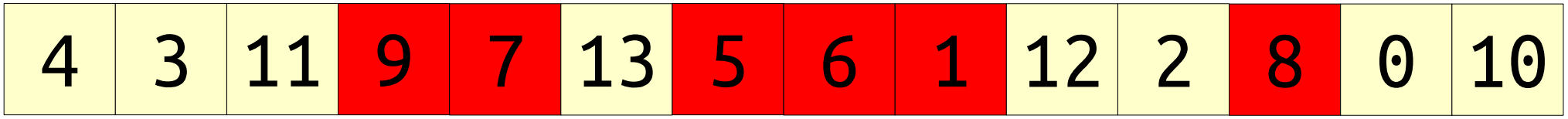
Goal: Find the length of the longest increasing subsequence of this sequence.

# A Sample Problem



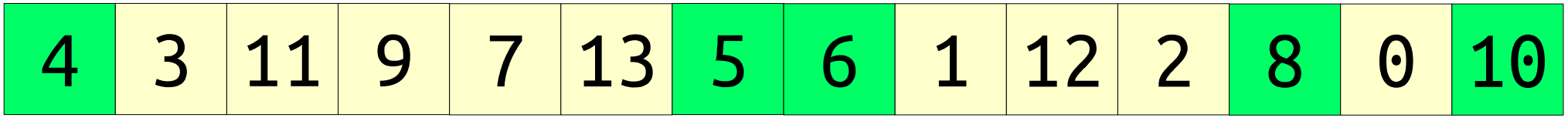

Goal: Find the length of the longest increasing subsequence of this sequence.

# A Sample Problem




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

# A Sample Problem



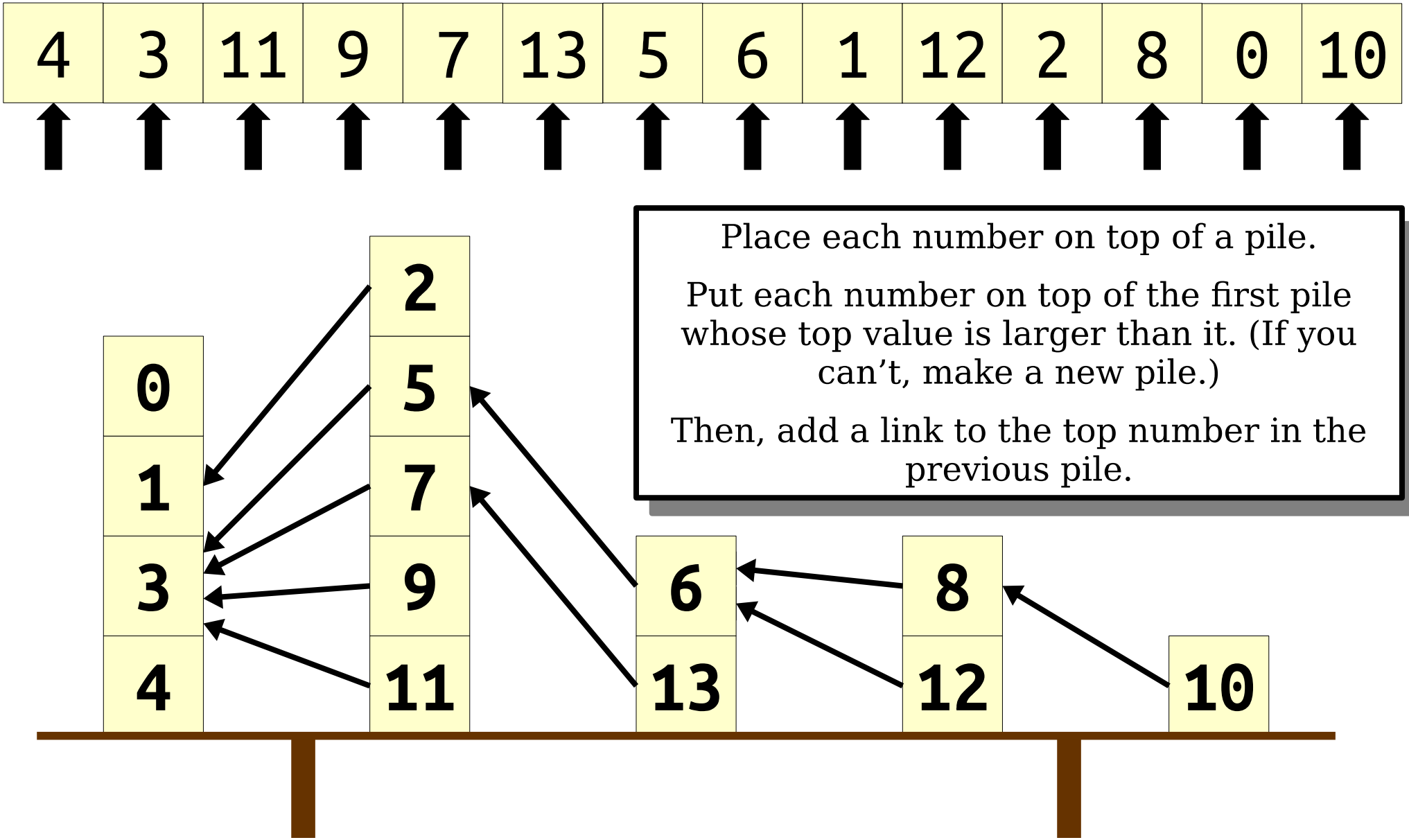

Goal: Find the length of the longest increasing subsequence of this sequence.

# 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 defines a subsequence.)
- Checking all of them to find the longest increasing subsequence will take time $O(n \cdot 2^n)$.
- ***Fact:*** the age of the universe is about $4.3 \times 10^{26}$ nanoseconds. That's about $2^{85}$ nanoseconds.
- Practically speaking, this algorithm doesn't terminate if you give it an input of size 100 or more.

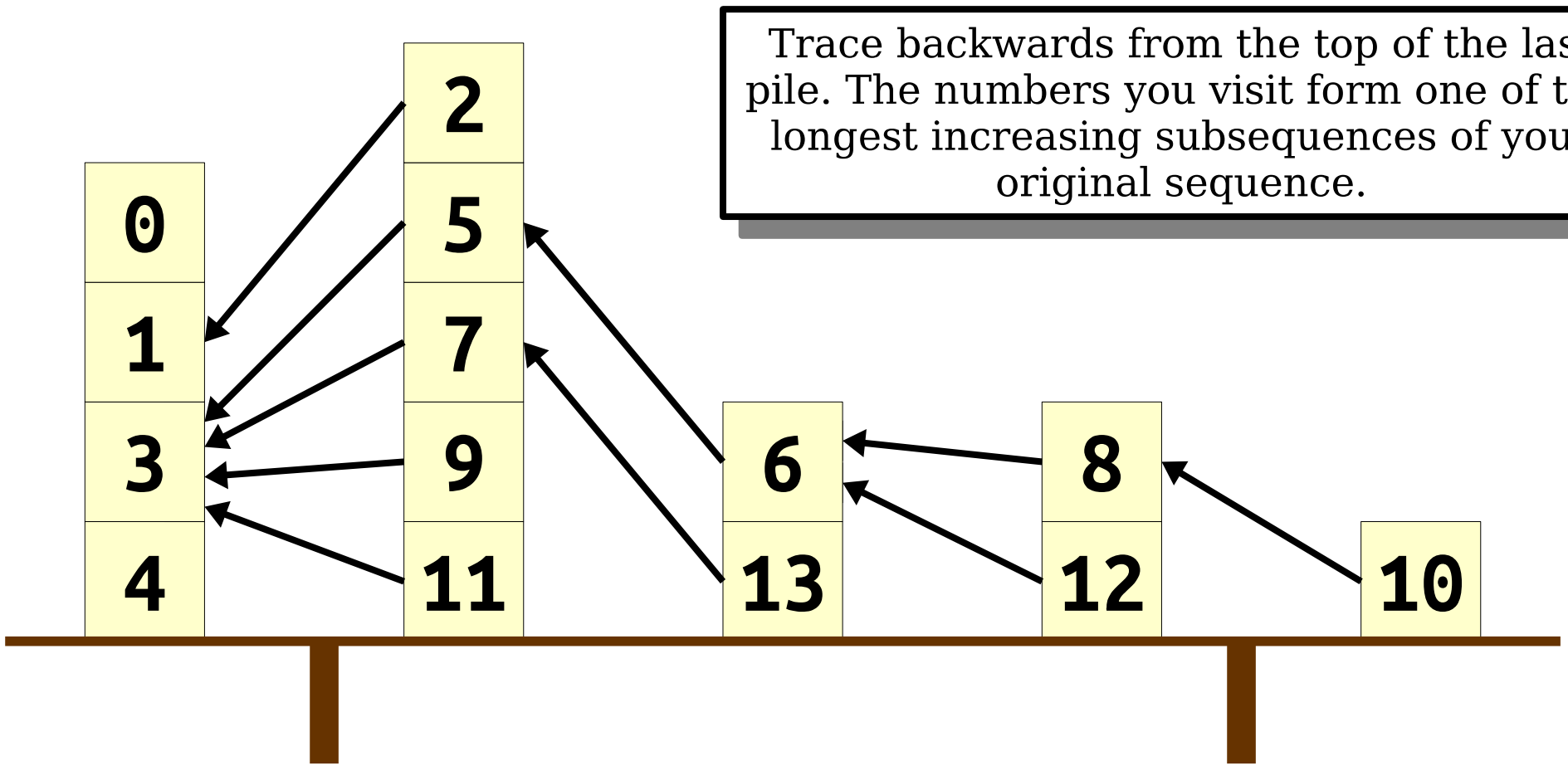
# A Different Approach

# Patience Sorting



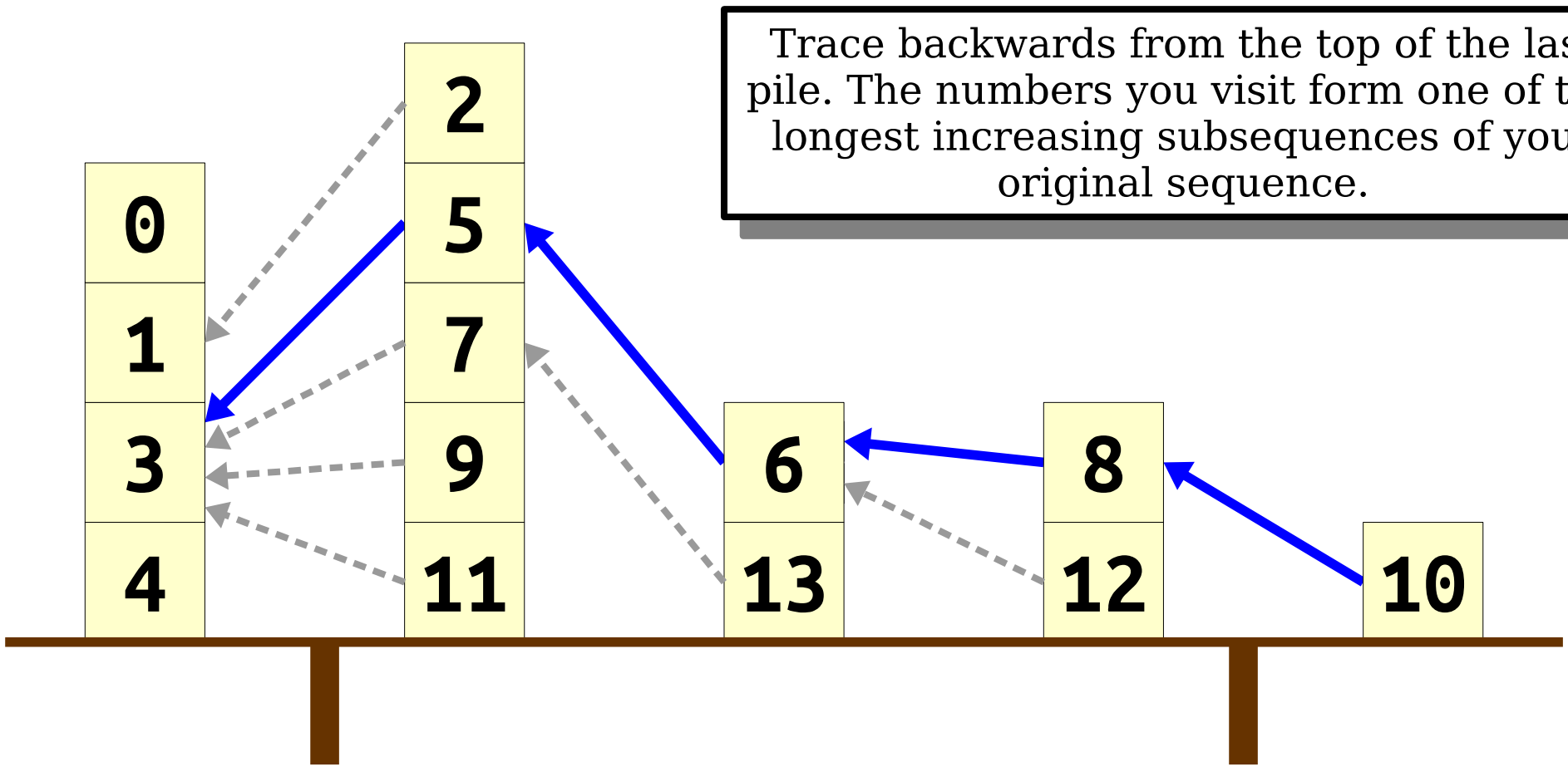
# Patience Sorting

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



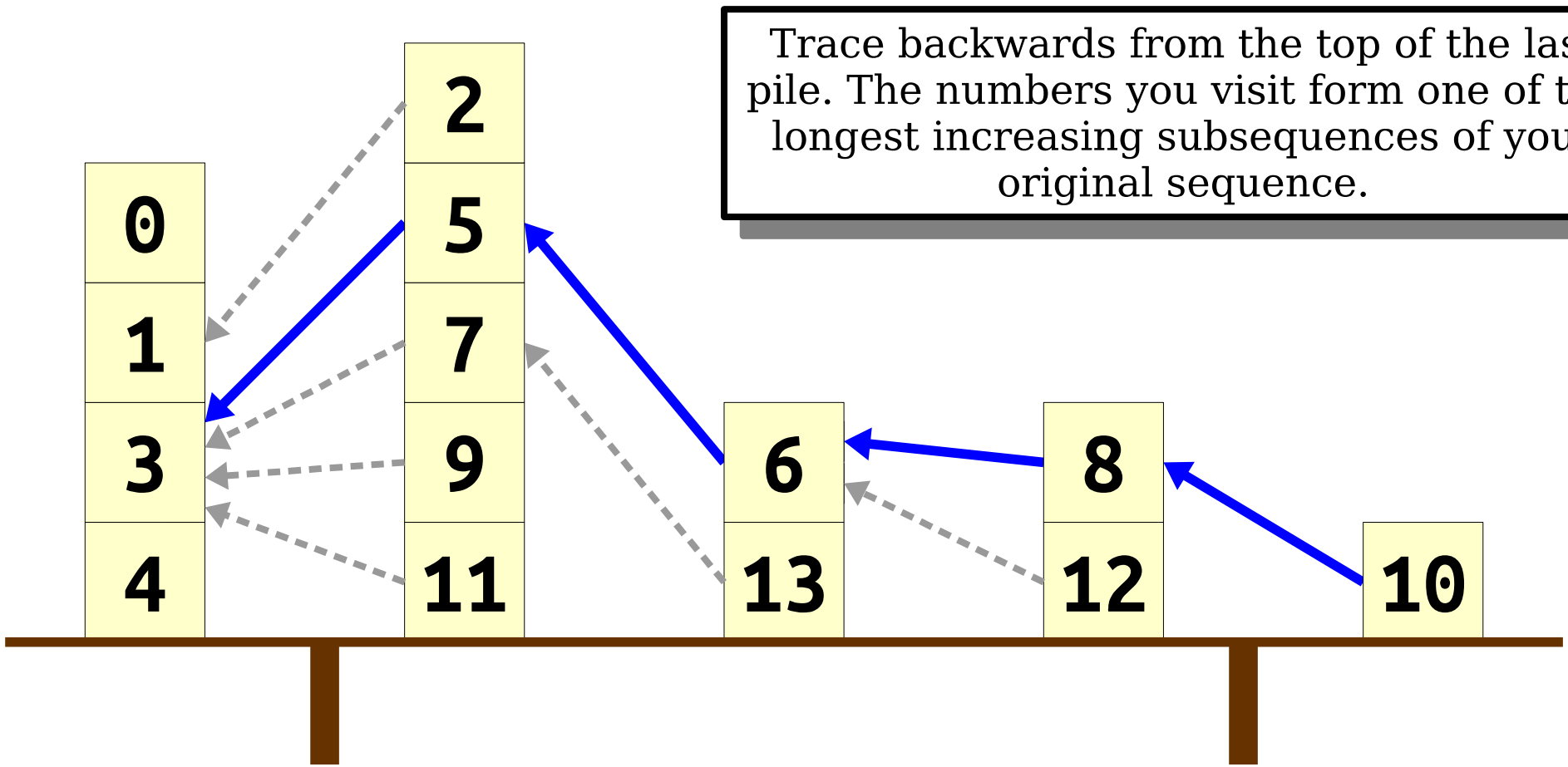
# Patience Sorting

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



# Patience Sorting

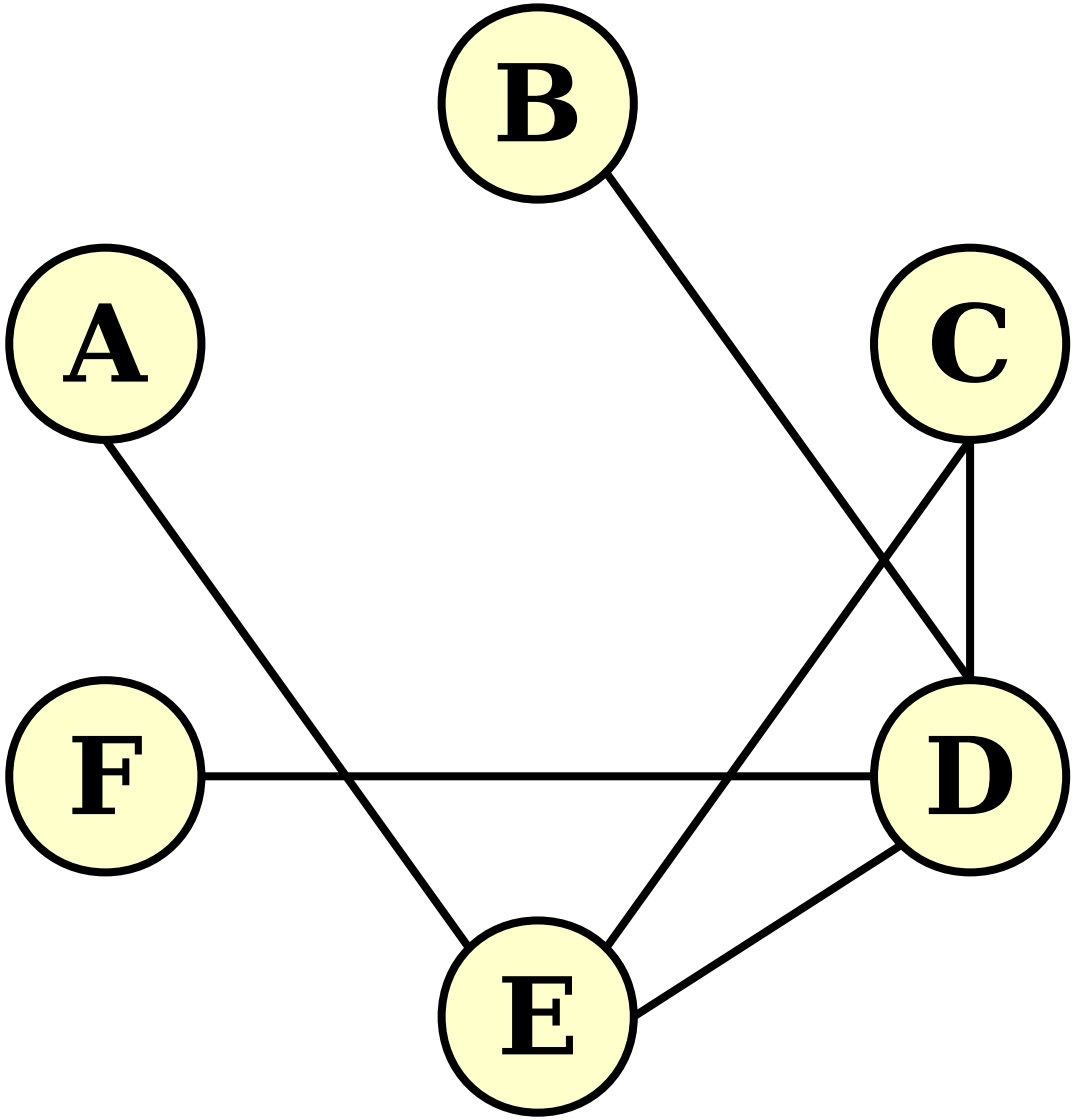
| 4 | 3 | 11 | 9 | 7 | 13 | 5 | 6 | 1 | 12 | 2 | 8 | 0 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|



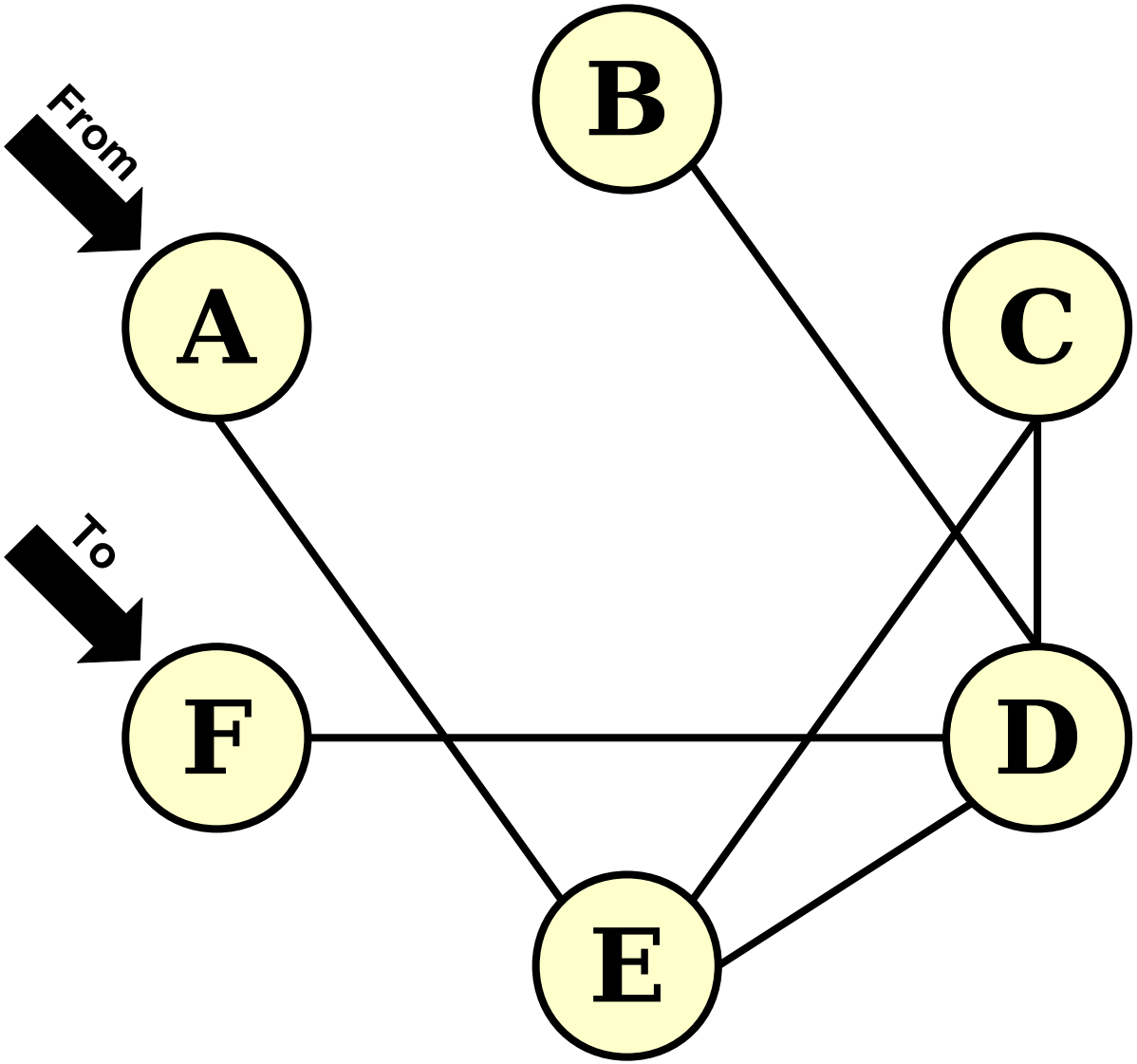
# Longest Increasing Subsequences

- ***Theorem:*** There is an algorithm that can find the longest increasing subsequence of an array in time $O(n^2)$.
  - It's the previous ***patience sorting*** algorithm, with some clever implementation tricks.
- This algorithm works by exploiting particular aspects of how longest increasing subsequences are constructed. It's not immediately obvious that it works correctly.

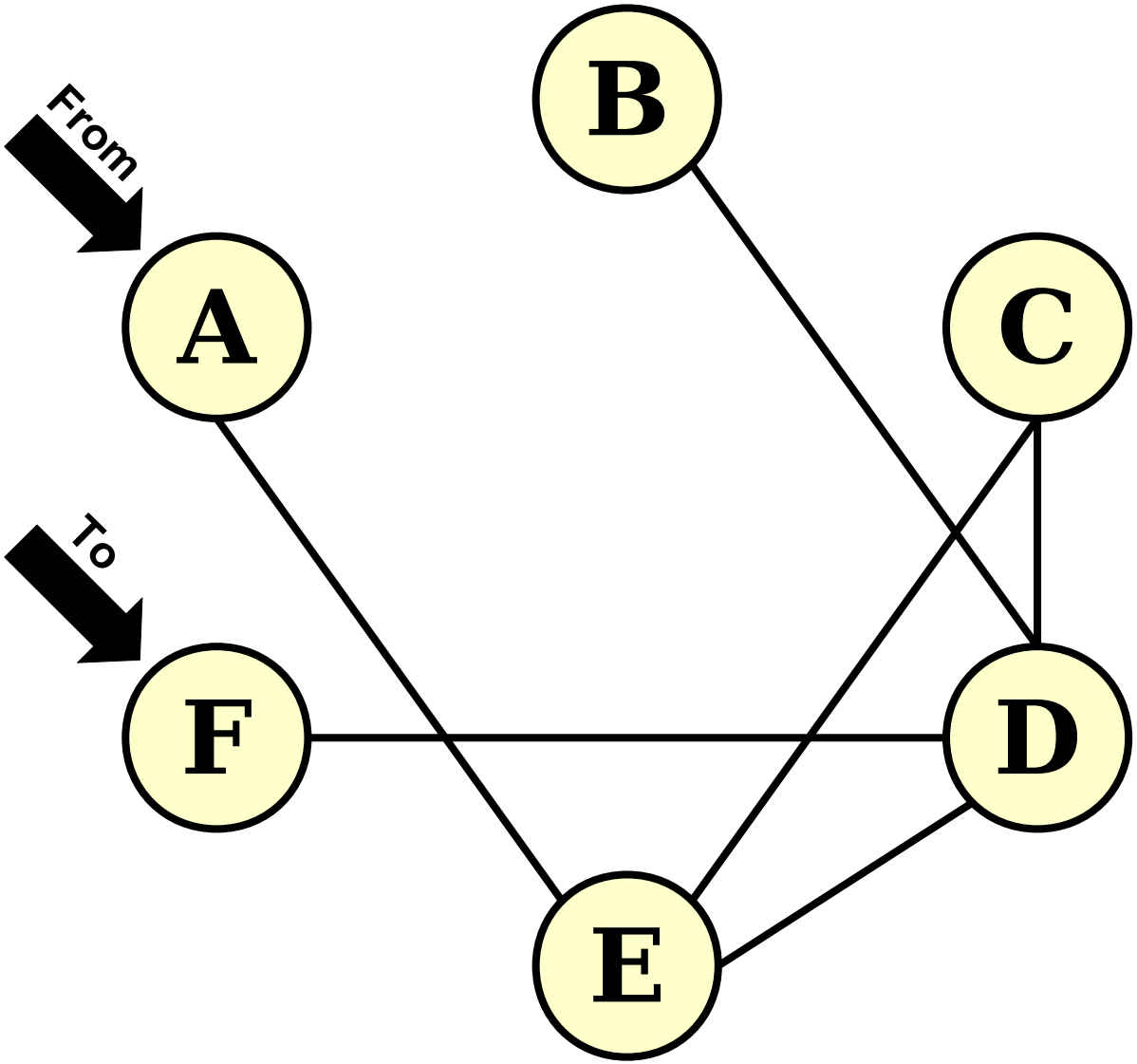
# Another Problem



# Another Problem



# Another Problem




Goal: Determine the length of the shortest path from **F** to **A** 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

- **Pop Quiz!** How else could we find a shortest path?

# Shortest Paths

- ***Theorem:*** It's possible to find the shortest path between two nodes in an $n$-node, $m$-edge graph in time $O(m + n)$.
- ***Proof idea:*** Use breadth-first search!
- This scales nicely!
- The algorithm is a bit nuanced. It uses some specific properties of shortest paths and the proof of correctness is 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 in 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

Which of the following are considered efficient runtimes?

| | |
|---|---|
| *1* | $n^2 - 3n + 17$ |
| 2 | $n \log n$ |
| 3 | $n^{1,000,000,000}$ |
| 4 | $n^n$ |
| 5 | $n!$ |
| 6 | $2^n$ |
| 7 | $1.0000001^n$ |
| 8 | $10^{500}$ |

Go to
PollEv.com/cs103spr25

# The Cobham-Edmonds Thesis

Which of the following are considered efficient runtimes?

| | | | |
|---|---|---|---|
| 1 | $n^2 - 3n + 17$ | ✔ | This is a polynomial in $n$. |
| 2 | $n \log n$ | ✔ | Bounded by $n^2$. |
| 3 | $n^{1,000,000,000}$ | ✔ | This is a polynomial in $n$. |
| 4 | $n^n$ | ✘ | Eventually bigger than $n^k$ for all $k$. |
| 5 | $n!$ | ✘ | Eventually bigger than $n^k$ for all $k$. |
| 6 | $2^n$ | ✘ | Eventually bigger than $n^k$ for all $k$. |
| 7 | $1.0000001^n$ | ✘ | Eventually bigger than $n^k$ for all $k$. |
| 8 | $10^{500}$ | ✔ | $10^{500} = 10^{500}\ n^0$ is a polynomial in $n$. |

# 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, then another, 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**

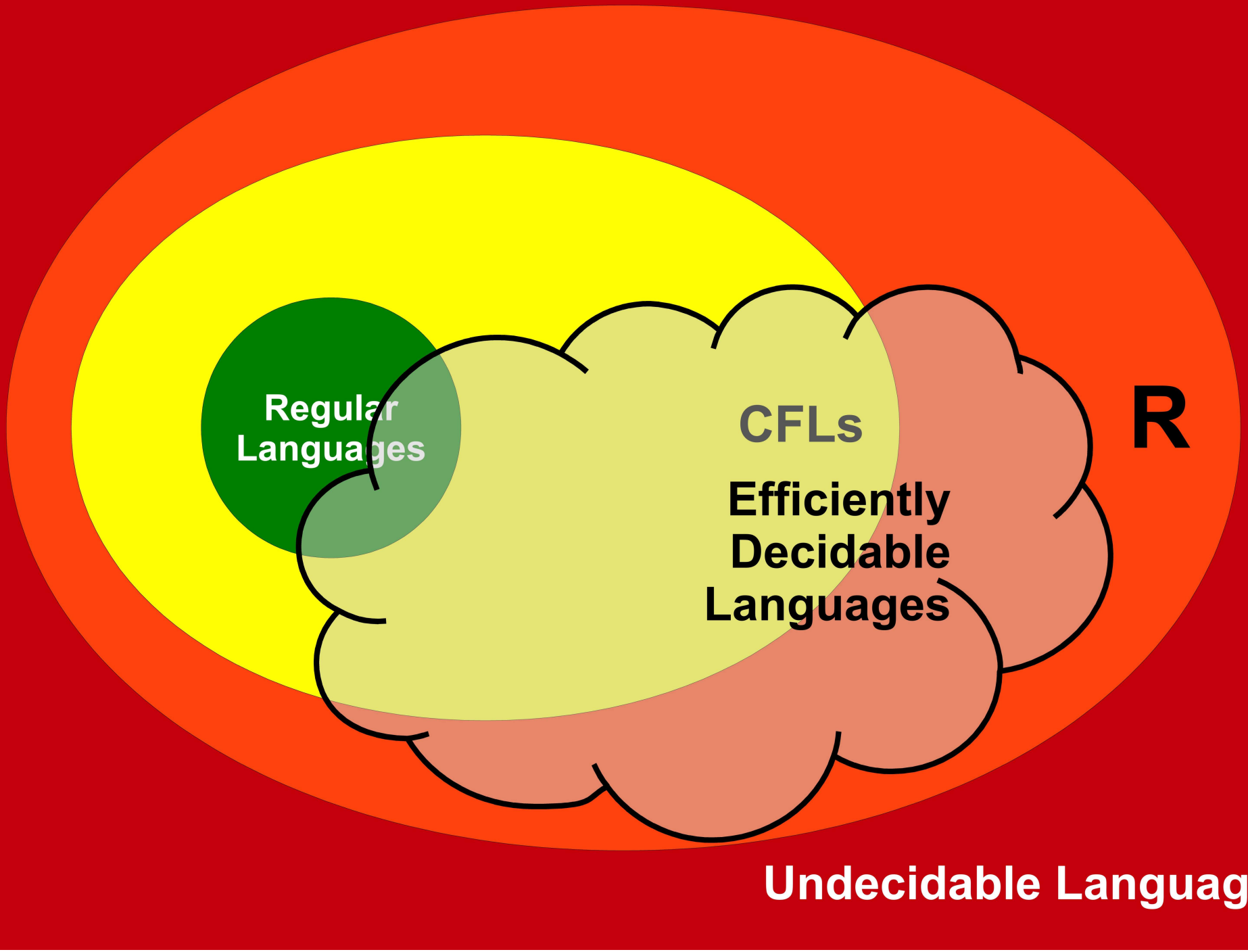
- The ***complexity class* P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

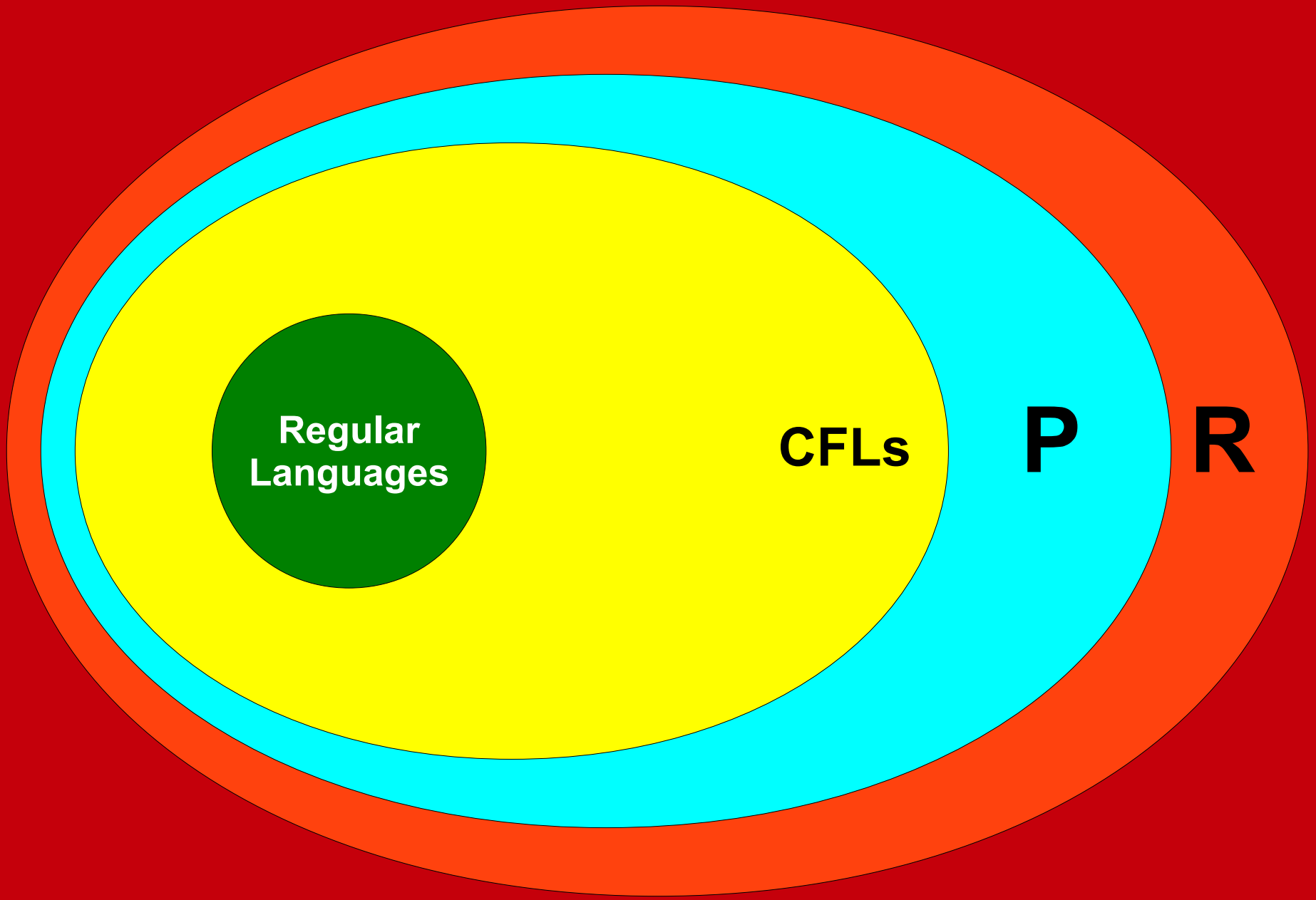
$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$

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

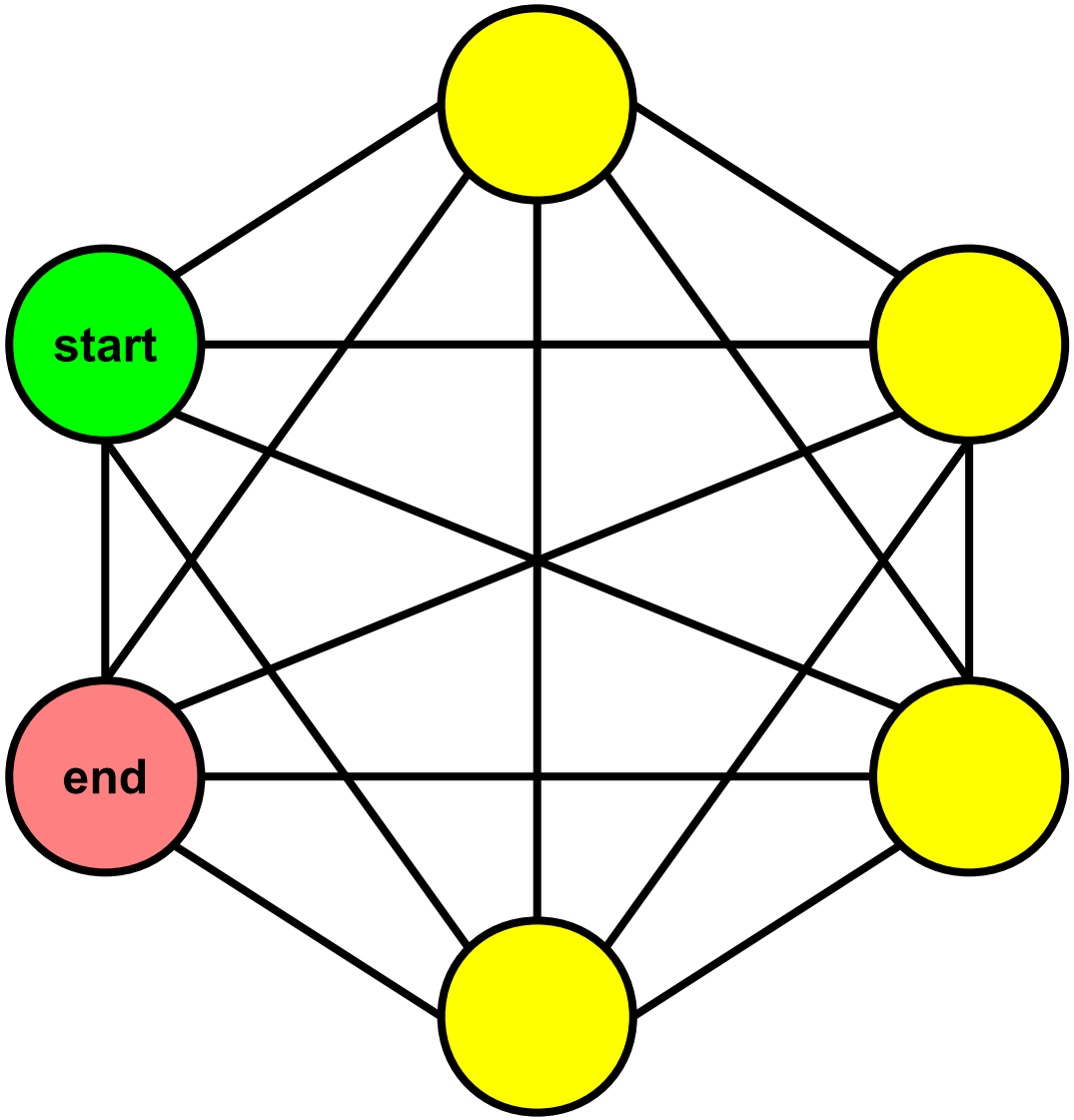
# Examples of Problems in **P**

- All regular languages are in **P**.
  - All have linear-time TMs.
- All CFLs are in **P**.
  - Requires a more nuanced argument (the *CYK algorithm* or *Earley's algorithm*).
- And a *ton* of other problems are in **P** as well.
  - Curious? Take CS161!


Regular Languages
CFLs
Efficiently Decidable Languages
R
Undecidable Languages


Regular Languages
CFLs
P
R
Undecidable Languages

What *can't* you do in polynomial time?




How many paths are there from the start node to the end node?

{ 

, 

, 

, 

}

How many subsets of this set are there?

# An Interesting Observation

- There are (at least) exponentially many objects of each of the preceding types.
- However, each of those objects is not very large.
  - Each simple path has length no longer than the number of nodes in the graph.
  - Each subset of a set has no more elements than the original set.
- This brings us to our next topic...

What if you need to search a large space for a single object?

# Verifiers – Again

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 6 | | 1 | | |
| | | | | | 3 | | 5 | 2 |
| 3 | | | 1 | | 5 | 9 | | 7 |
| 6 | | 5 | | 3 | | 8 | | 9 |
| | 1 | | | | | | 2 | |
| 8 | | 2 | | 1 | | 5 | | 4 |
| 1 | | 3 | 2 | | 7 | | | 8 |
| 5 | 7 | | 4 | | | | | |
| | | 4 | | 8 | | 7 | | |

Does this Sudoku problem have a solution?

# Verifiers – Again

| 2 | 5 | 7 | 9 | 6 | 4 | 1 | 8 | 3 |
|---|---|---|---|---|---|---|---|---|
| 4 | 9 | 1 | 8 | 7 | 3 | 6 | 5 | 2 |
| 3 | 8 | 6 | 1 | 2 | 5 | 9 | 4 | 7 |
| 6 | 4 | 5 | 7 | 3 | 2 | 8 | 1 | 9 |
| 7 | 1 | 9 | 5 | 4 | 8 | 3 | 2 | 6 |
| 8 | 3 | 2 | 6 | 1 | 9 | 5 | 7 | 4 |
| 1 | 6 | 3 | 2 | 5 | 7 | 4 | 9 | 8 |
| 5 | 7 | 8 | 4 | 9 | 6 | 2 | 3 | 1 |
| 9 | 2 | 4 | 3 | 8 | 1 | 7 | 6 | 5 |

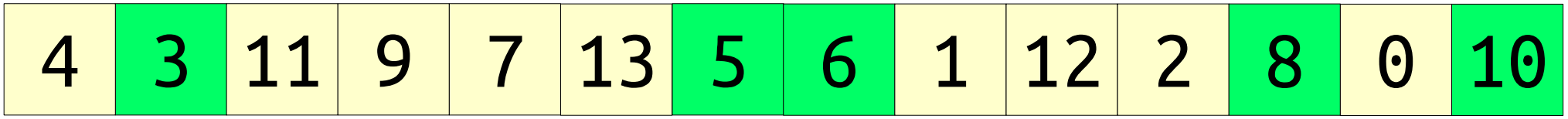
Does this Sudoku problem have a solution?

# Verifiers – Again

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

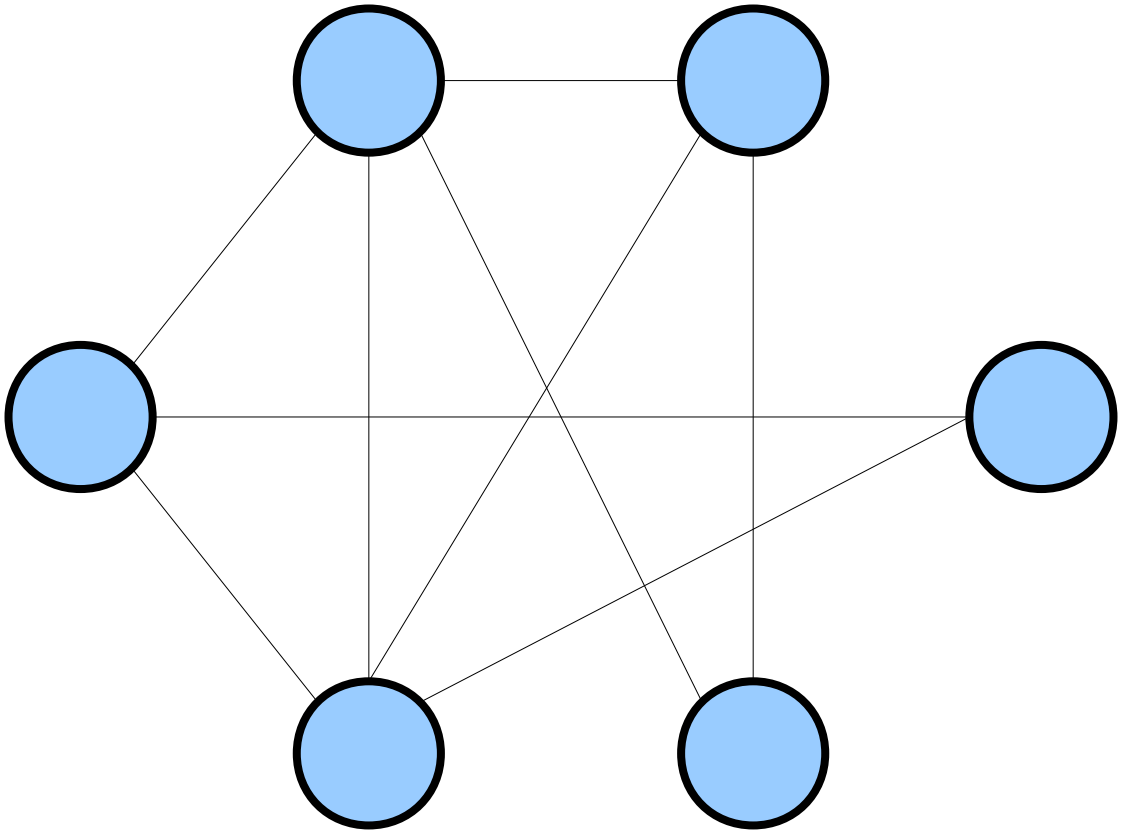
Is there an ascending subsequence of length at least 5?

# Verifiers – Again



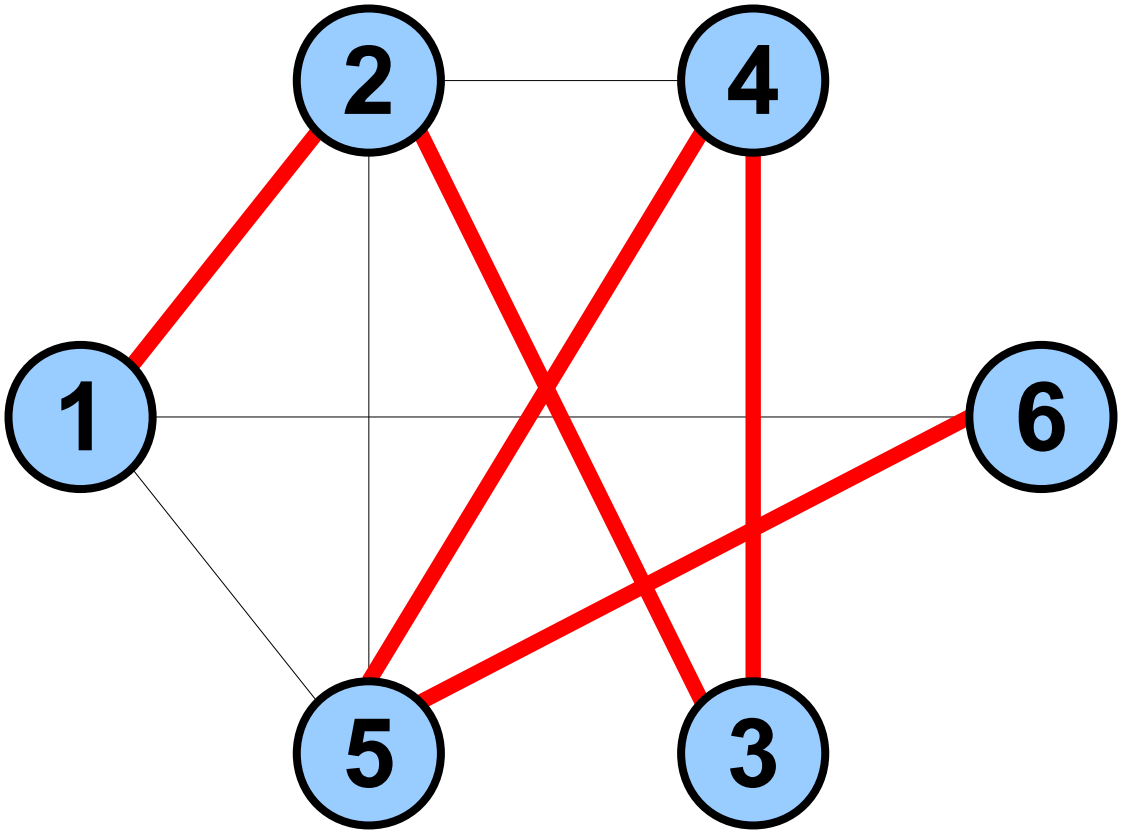

Is there an ascending subsequence of length at least 5?

# Verifiers – Again



Is there a path that goes through every node exactly once?

# Verifiers – Again




Is there a path that goes through every node exactly once?

# Verifiers

- Recall that a ***verifier*** for $L$ is a TM $V$ such that
  - $V$ halts on all inputs.
  - $w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*$. $V$ accepts $\langle w, c \rangle$.

# Polynomial-Time Verifiers

- A ***polynomial-time verifier*** for $L$ is a TM $V$ such that
  - $V$ halts on all inputs.
  - $w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*$. $V$ accepts $\langle w, c \rangle$.
  - $V$ runs "efficiently" (its runtime is $O(|w|^k)$ for some $k \in \mathbb{N}$).
  - All strings in $L$ have "short" certificates (their lengths are $O(|w|^r)$ for some $r \in \mathbb{N}$).

# The Complexity Class **NP**

- The complexity class **NP** (***nondeterministic polynomial time***) contains all problems that can be verified in polynomial time.
- Formally:

$$\textbf{NP} = \{ L \mid \text{There is a polynomial-time verifier for } L \}$$

- The name **NP** comes from another way of characterizing **NP**. If you introduce *nondeterministic Turing machines* and appropriately define "polynomial time," then **NP** is the set of problems that an NTM can solve in polynomial time.
- ***Useful fact:*** **NP** ϖ**R**.
  - ***Proof idea:*** If $L \in \textbf{NP}$, all strings in $L$ have "short" certificates. Therefore, we can just try all possible "short" certificates and see if any of them work. (Showing **NP** is a strict subset of **R** requires some more advanced techniques.)

$$\mathbf{P} = \{ L \mid \text{there is a polynomial-time decider for } L \}$$

$$\mathbf{NP} = \{ L \mid \text{there is a polynomial-time verifier for } L \}$$

$$\mathbf{R} = \{ L \mid \text{there is a } \cancel{\text{polynomial-time}} \text{ decider for } L \}$$

$$\mathbf{RE} = \{ L \mid \text{there is a } \cancel{\text{polynomial-time}} \text{ verifier for } L \}$$

We know that **R ≠ RE**.

So does that mean **P ≠ NP**?

And now…

The

***Biggest Question***

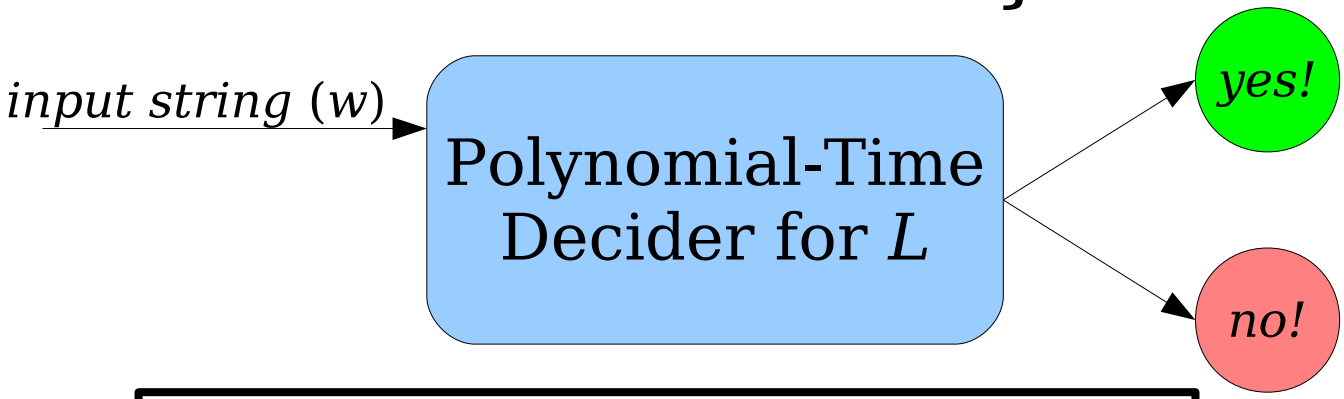
in

***Theoretical Computer Science***

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}$$

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

**NP** = { $L$ | There is a polynomial-time verifier for $L$ }

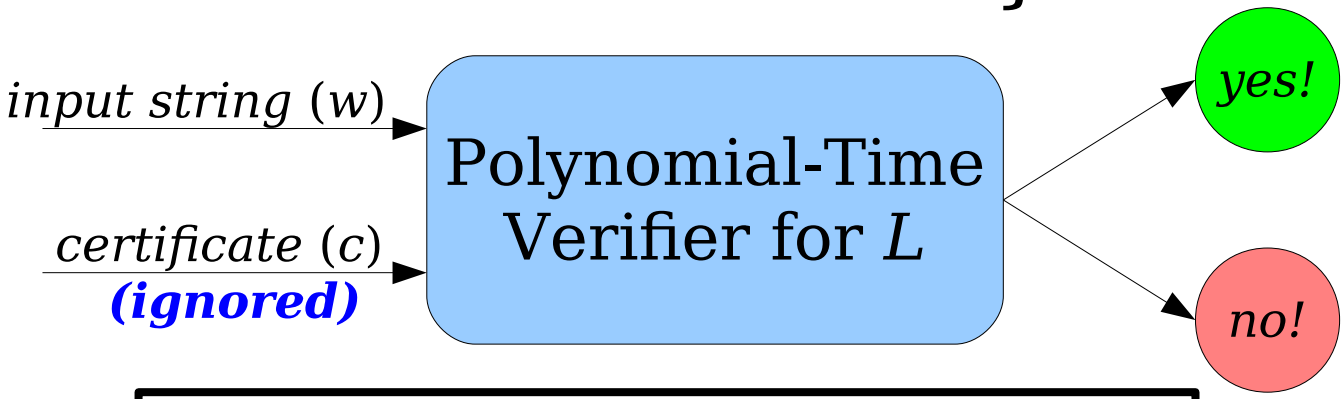



```
bool solveProblemL(string w) {

    do some work;
    return the answer;
}
```

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

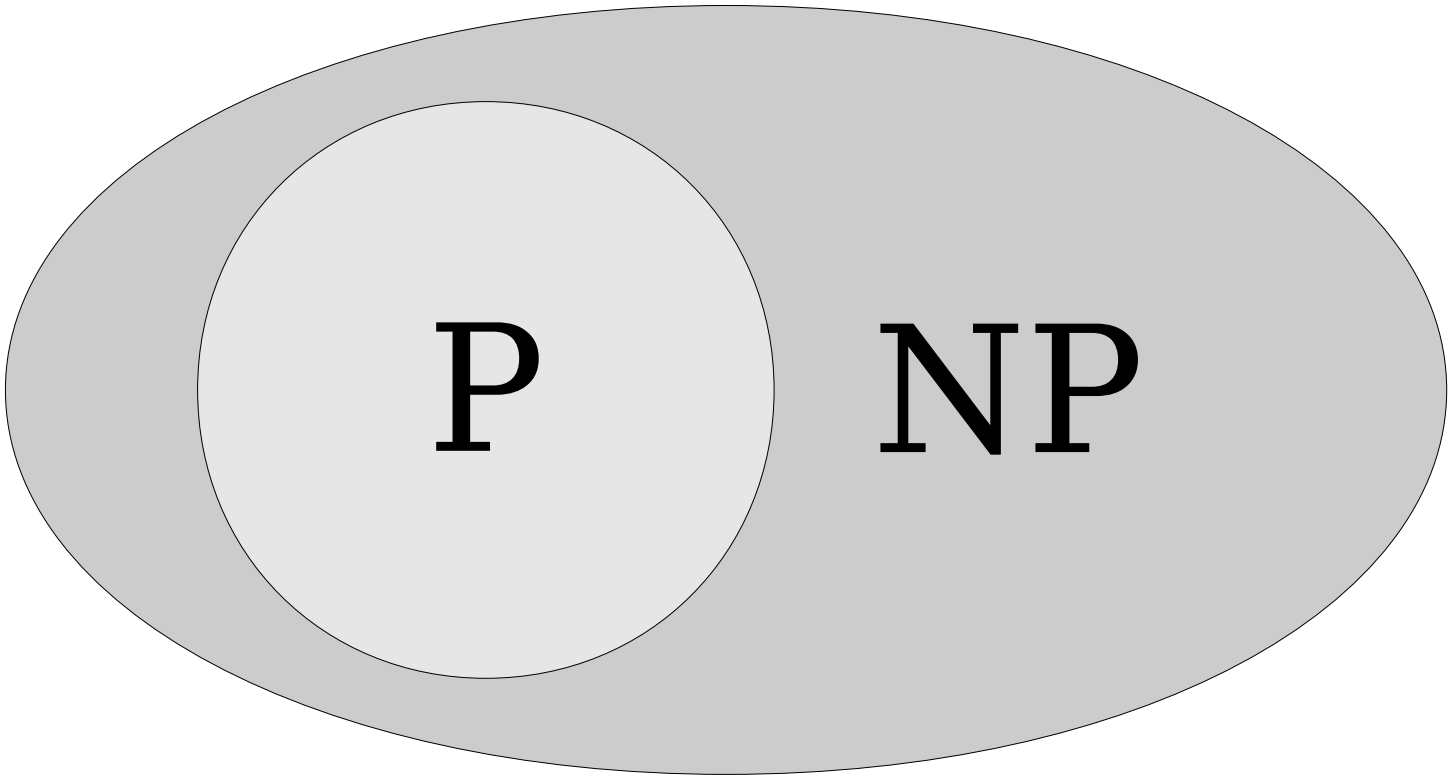
**NP** = { $L$ | There is a polynomial-time verifier for $L$ }



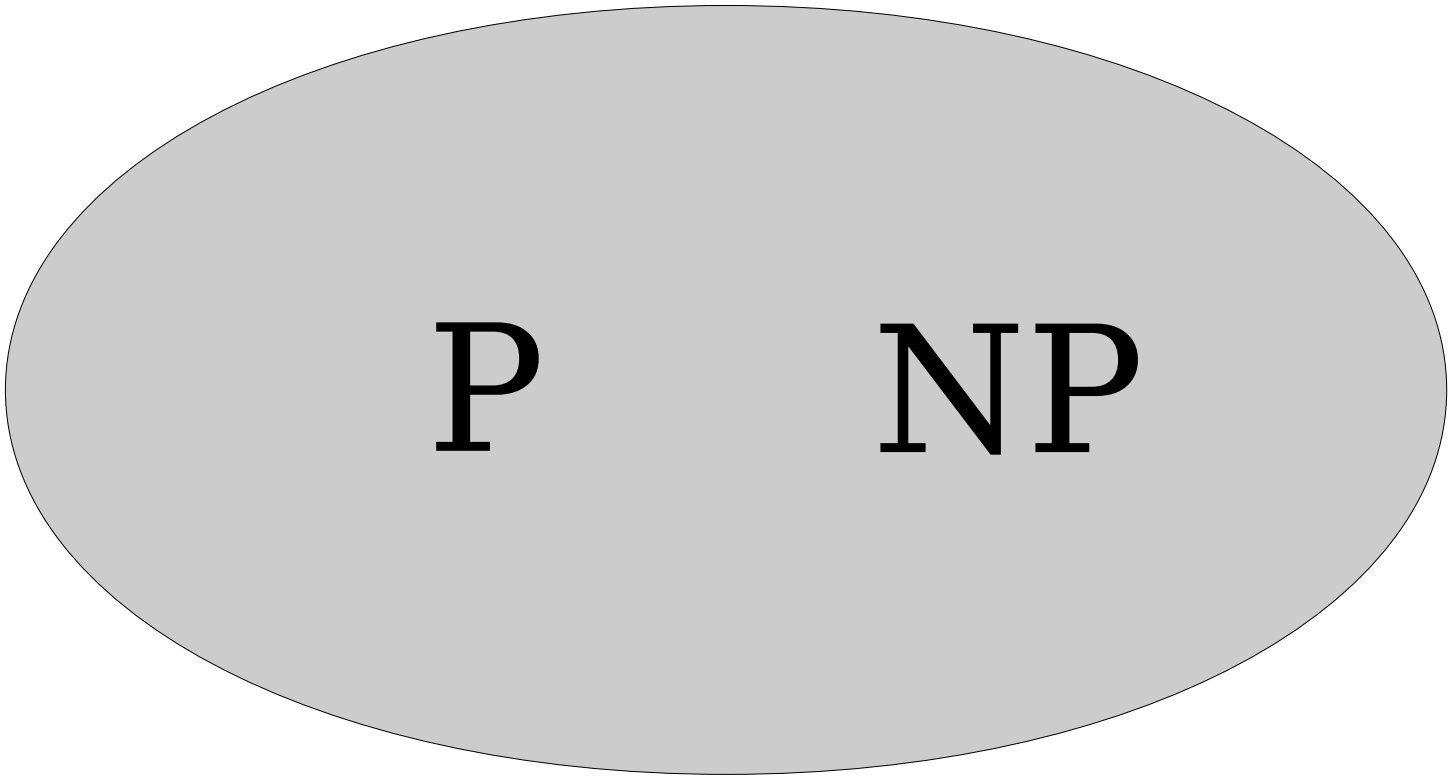

```
bool solveProblemL(string w, string c) {
    /* don't even look at c */
    do some work;
    return the answer;
}
```

**P ⊆ NP**

# Which Picture is Correct?



# Which Picture is Correct?



# **P** ≟ **NP**

- The **P** ≟ **NP** question is the most important open question in theoretical computer science.
- With the verifier definition of **NP**, one way of phrasing this question is

  *If a solution to a problem can be **checked** efficiently, can that problem also be **solved** efficiently?*

- An answer either way will give fundamental insights into the nature of computation.

# Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem).
  - Determining whether a simple DNA strand exists that multiple gene sequences could be a part of (shortest common supersequence).
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation).
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling).
  - ***And many more***.
- If **P = NP**, ***all*** of these problems have efficient solutions.
- If **P ≠ NP**, ***none*** of these problems have efficient solutions.

# Why This Matters

- If **P = NP**:
  - A huge number of seemingly difficult problems could be solved efficiently.
  - Our capacity to solve many problems will scale well with the size of the problems we want to solve.
- If **P ≠ NP**:
  - Enormous computational power would be required to solve many seemingly easy tasks.
  - Our capacity to solve problems will fail to keep up with our curiosity.

# What We Know

- Resolving **P** ≟ **NP** has proven ***extremely difficult***.
- In the past 50 years:
  - Not a single correct proof either way has been found.
  - Many types of proofs have been shown to be insufficiently powerful to determine whether **P** ≟ **NP**.
  - A majority of computer scientists believe **P ≠ NP**, but it isn't an overwhelming majority.
- Interesting read: Interviews with leading thinkers about **P** ≟ **NP**:
  - https://www.cs.umd.edu/~gasarch/papers/poll.pdf

# The Million-Dollar Question


The Clay Mathematics Institute has offered a ***$1,000,000 prize*** to anyone who proves or disproves **P = NP**.

“My hunch is that [**P** ̛P **NP**] will be solved by a young researcher who is not encumbered by too much conventional wisdom about how to attack the problem.”

– Prof. Richard Karp

***(The guy who first popularized the* P ̛PNP *problem.)***

What do we know about **P** ʿP **NP**?

# Adapting our Techniques

$$\mathbf{P} = \{ L \mid \text{there is a polynomial-time decider for } L \}$$

$$\mathbf{NP} = \{ L \mid \text{there is a polynomial-time verifier for } L \}$$

$$\mathbf{R} = \{ L \mid \text{there is a } \cancel{\text{polynomial-time}} \text{ decider for } L \}$$

$$\mathbf{RE} = \{ L \mid \text{there is a } \cancel{\text{polynomial-time}} \text{ verifier for } L \}$$

We know that **R ≠ RE**.

So does that mean **P ≠ NP**?

# A Problem

- The **R** and **RE** languages correspond to problems that can be decided and verified, *period*, without any time bounds.
- To reason about what's in **R** and what's in **RE**, we used two key techniques:
  - ***Universality***: TMs can simulate other TMs.
  - ***Self-Reference***: TMs can get their own source code.
- Why can't we just do that for **P** and **NP**?

***Theorem (Baker-Gill-Solovay):*** Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

***Proof:*** Take CS154!

# Announcements

- Problem Set 7 was due at 5:30 PM. Solutions are available on the course website.

***Congratulations – you're done with CS103 problem sets!***

- Problem Set 6 has been graded, I'm calibrating some questions with the CAs. Grades will be released on Gradescope tonight.

***Please evaluate this course on Axess.***

Your feedback really makes a difference.

# Final Exam Logistics

- Our final exam will be on Saturday, August 16th from 7:00 – 10:00 PM in Hewlett 201 (same as the lectures and the midterm).
- Exam is the same format as the midterm: 3 hours, closed-book, 1 page of notes allowed.
- If you have OAE accommodations, you should have heard from us already about exam room and time logistics.
- The exam is cumulative and covers Lectures 00 – 19 as well as PS1 – PS6.
- ***Best of luck on the exam – you've got this!***

Take a minute to reflect on your journey.

Set Theory
Power Sets
Cantor’s Theorem
Direct Proofs
Parity
Proof by Contrapositive
Proof by Contradiction
Modular Congruence
Propositional Logic
First-Order Logic
Logic Translations
Logical Negations
Propositional Completeness
Vacuous Truths
Perfect Squares
Tournaments
Functions
Injections
Surjections
Involutions
Monotone Functions
Bijections
Cardinality
Graphs
Connectivity
Independent Sets
Vertex Covers
Graph Complements
Dominating Sets
Bipartite Graphs
The Pigeonhole Principle
Ramsey Theory
Mathematical Induction
Loop Invariants
Complete Induction
Formal Languages
DFAs
Regular Languages
Closure Properties
NFAs
Subset Construction
Kleene Closures
Regular Expressions
State Elimination
Distinguishability
Myhill-Nerode Theorem
Nonregular Languages
Context-Free Grammars
Brzozowski’s Theorem
Turing Machines
Church-Turing Thesis
TM Encodings
Universal Turing Machines
Self-Reference
Decidability
Recognizability
Self-Defeating Objects
Undecidable Problems
The Halting Problem
Verifiers
Diagonalization Language
Complexity Class **P**
Complexity Class **NP**
**P** $\stackrel{?}{=}$ **NP** Problem

You’ve done more than just check
a bunch of boxes off a list.

You’ve given yourself the foundation to tackle problems from all over computer science.

**From CS255**

A **Shannon cipher** is a pair $\mathcal{E} = (E, D)$ of functions.

- The function $E$ (the **encryption function**) takes as input a **key** $k$ and a **message** $m$ (also called a **plaintext**), and produces as output a **ciphertext** $c$. That is,

$$c = E(k, m),$$

and we say that $c$ is the **encryption of** $m$ **under** $k$.

- The function $D$ (the **decryption function**) takes as input a key $k$ and a ciphertext $c$, and produces a message $m$. That is,

$$m = D(k, c),$$

and we say that $m$ is the **decryption of** $c$ **under** $k$.

- We require that decryption "undoes" encryption; that is, the cipher [illegible] **correctness property**: for all keys $k$ and all messages $m$, we have

$$D(k,\ E(k,\ m)\ ) = m.$$

Kinda sorta like a left inverse!

To be slightly more formal, let us assume that $\mathcal{K}$ is the set of all keys (the **key space**), $\mathcal{M}$ is the set of all messages (the **message space**), and that $\mathcal{C}$ is the set of all ciphertexts (the **ciphertext space**). With this notation, we can write:

$$E : \mathcal{K} \times \mathcal{M} \to \mathcal{C},$$
$$D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}.$$

Also, we shall say that $\mathcal{E}$ is **defined over** $(\mathcal{K}, \mathcal{M}, \mathcal{C})$.

# Strong triadic closure

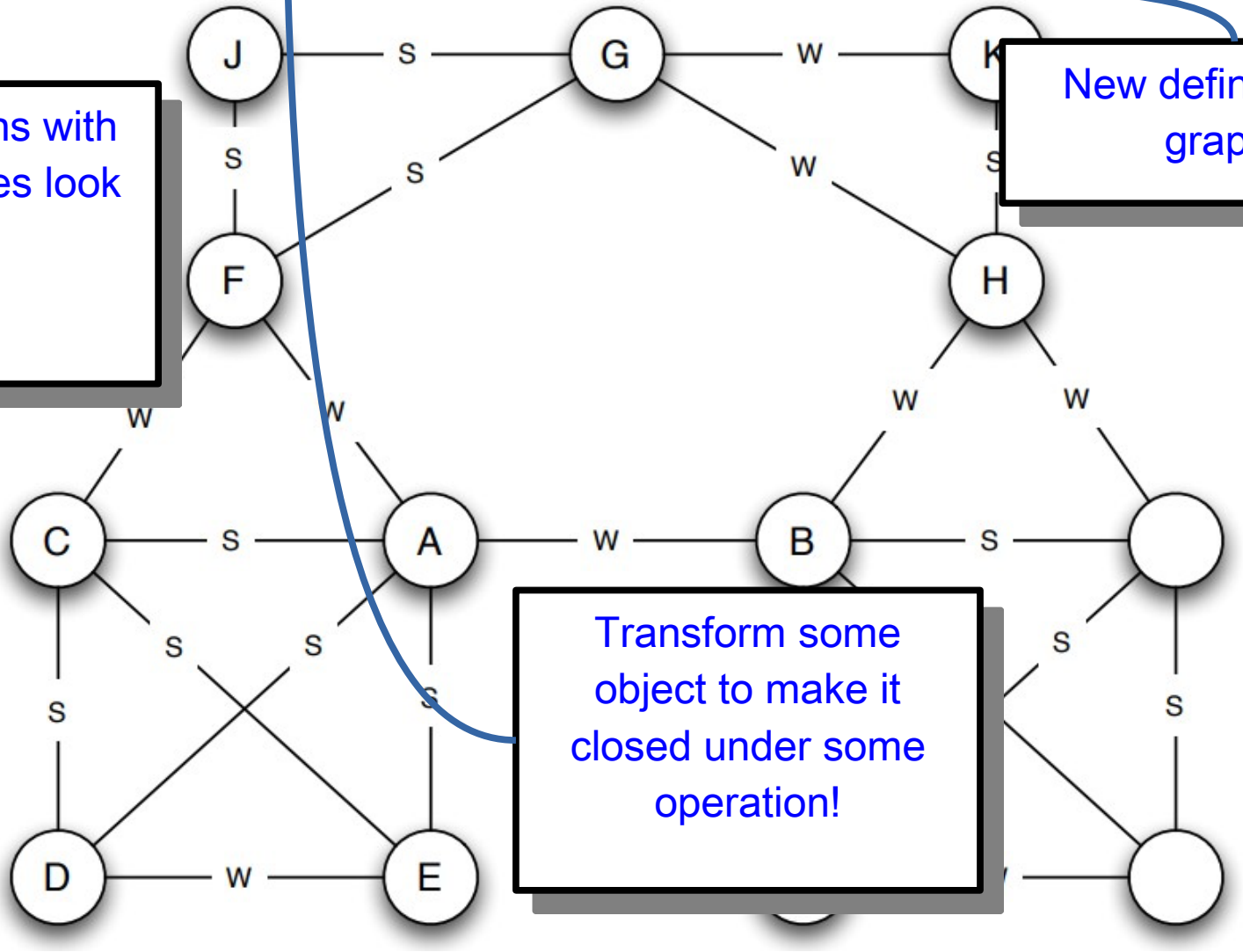
*From CS124*

If a node Q has two strong ties to nodes Y and Z, there is an edge between Y and Z

What do graphs with these properties look like?

New definitions on graphs!

Transform some object to make it closed under some operation!



*From CS124*

# Tokenization in NLTK

Bird, Loper and Klein (2009), *Natural Language Processing with Python*. O'Reilly

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

It's a big regex!

Describing the world in set theory!

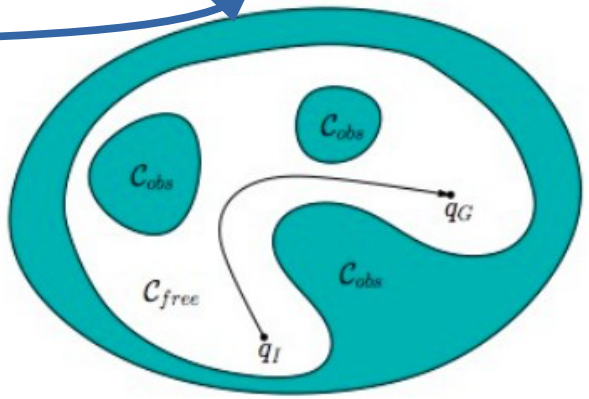
***From CS237A***

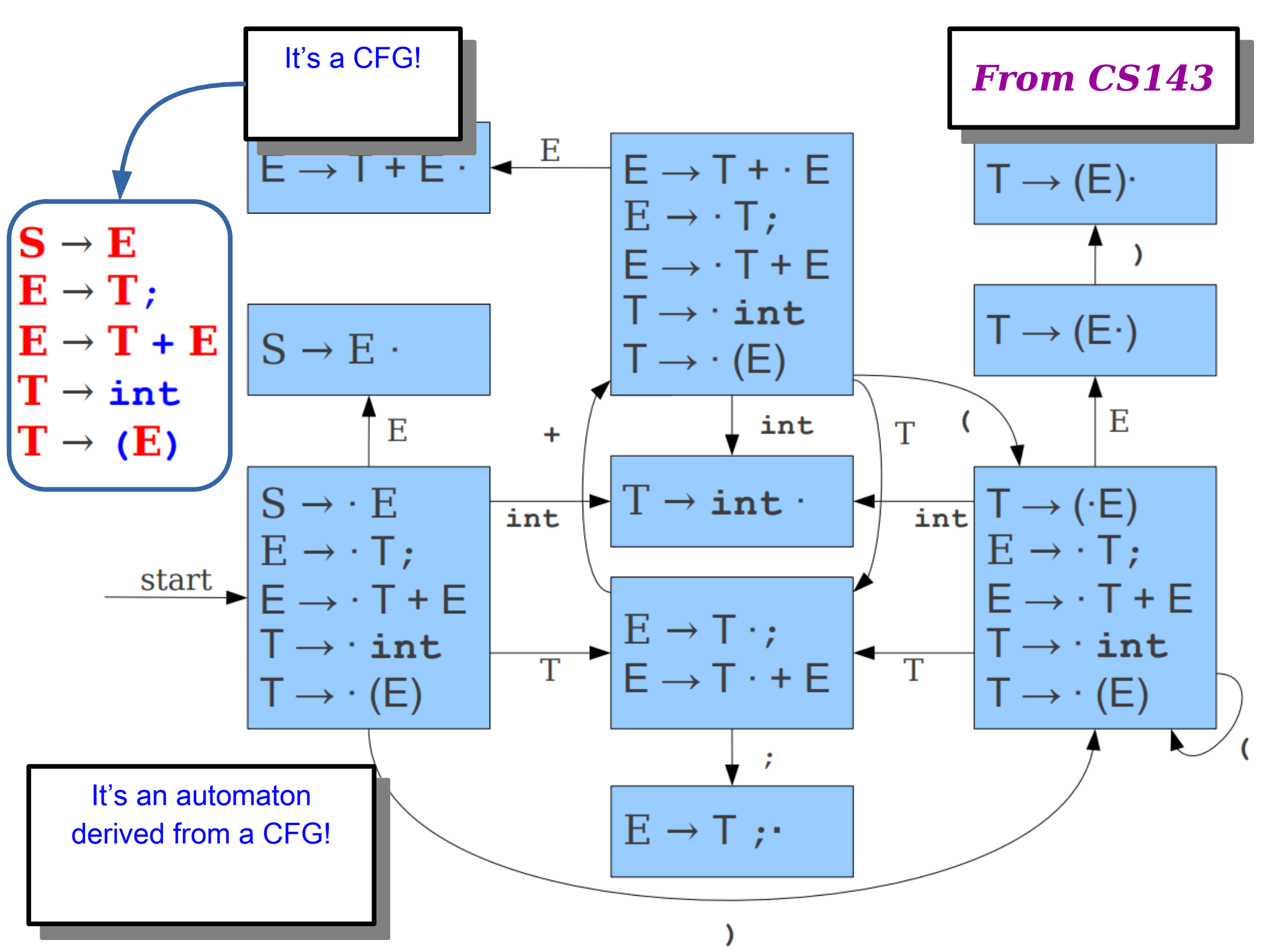
# Plan[illegible]space

- Let $R(q) \subset W$ denote set of points in the world occupied by robot when in configuration $q$
- Robot in collision $\Leftrightarrow R(q) \cap O \neq \emptyset$
- Accordingly, *free* space is defined as: $C_{free} = \{q \in C | R(q) \cap O = \emptyset\}$

- Path planning problem in *C*-space: compute a **continuous** path: $\tau: [0,1] \rightarrow C_{free}$, with $\tau(0) = q_I$ and $\tau(1) = q_G$

Model paths as functions!




It's a CFG!
From CS143
S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
T → (E·)
S → E ·
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → int ·
E → T ·;
E → T · + E
E → T ;·
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
E
+
int
T
(
)
;
It's an automaton
derived from a CFG!

# Search problems

*From CS221*

**Definition: search problem**

$\text{States}$: the set of states

$s_{\text{start}} \in \text{States}$: starting state

$\text{Actions}(s)$: possible actions from state $s$

$\text{Succ}(s, a)$: where we end up if take action $a$ in state $s$

$\text{Cost}(s, a)$: cost for taking action $a$ in state $s$

$\text{IsEnd}(s)$: whether at end

- $\text{Succ}(s, a) \Rightarrow T(s, a, s')$
- $\text{Cost}(s, a) \Rightarrow \text{Reward}(s, a, s')$

It's a DFA!

pronounced "big-oh of …" or sometimes "oh of …"

**From CS161**

# O(…) means an upper bound

- Let T(n), g(n) be functions of positive integers.
  - Think of T(n) as being a runtime: positive and increasing in n.

- We say "T(n) is O(g(n))" if g(n) grows at least as fast as T(n) as n gets large.

- Formally,

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$

It's FOL and functions!

**From CS224W**

- Graph $G(V, E)$ has **expansion $\alpha$**: if $\forall S \subseteq V$:

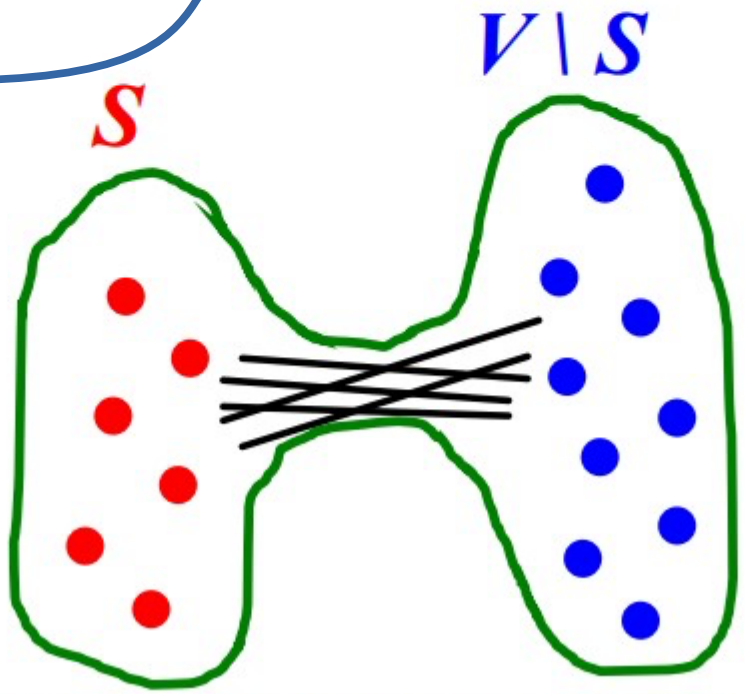
  $\#\text{ of edges leaving } S \geq \alpha \cdot \min(|S|, |V \setminus S|)$

- **Or equivalently:**

$$\alpha = \min_{S \subseteq V} \frac{\#\,edges\ leaving\ S}{\min(|S|, |V \setminus S|)}$$

Set difference and cardinality!

First-order definitions on graphs!



From CS242

# Typed lambda calculus

To understand the formal concept of a type system, we're going to extend our lambda calculus from last week (henceforth the "untyped" lambda calculus) with a notion of types (the "simply typed" lambda calculus). Here's the essentials of the language:

| | | |
|---|---|---|
| Type $\tau$ ::= | int | integer |
| \| | $\tau_1 \to \tau_2$ | function |
| Expression $e$ ::= | $x$ | variable |
| \| | $n$ | integer |
| \| | $e_1 \oplus e_2$ | binary operation |
| \| | $\lambda\ (x : \tau)\ .\ e$ | function |
| \| | $e_1\ e_2$ | application |
| Binop $\oplus$ ::= | $+ \mid - \mid * \mid /$ | |

It's a CFG!

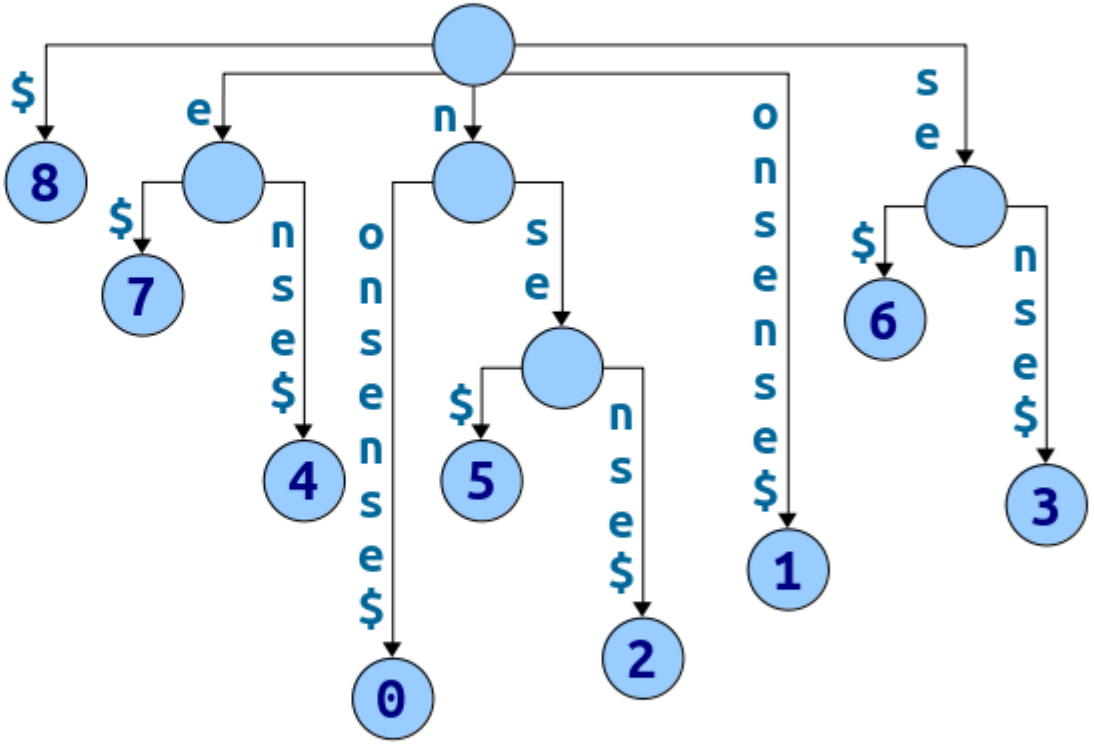
First, we introduce a language of types, indicated by the variable tau ($\tau$). A type is either an integer, or a function from an input type $\tau_1$ to an output type $\tau_2$. Then we extend our untyped lambda calculus with the same arithmetic language from the first lecture (numbers and binary operators)[4]. Usage of the language looks similar to before:

Definitions in terms of strings!

***From CS166***

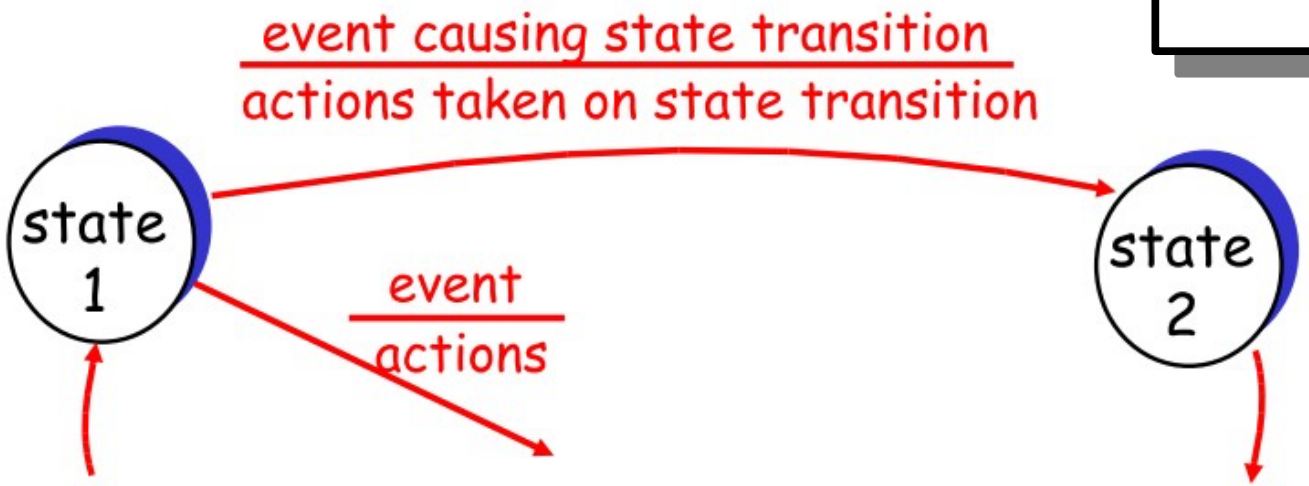
# The Anatomy of a Suffix Tree

- A ***branching word*** in $T\$$ is a string $\omega$ such that there are characters $a \neq b$ where $\omega a$ and $\omega b$ are substrings of $T\$$.
  - Edge case: the empty string is always considered branching.
- ***Theorem:*** The suffix tree for a string $T$ has an internal node for a string $\omega$ if and only if $\omega$ is a branching word in $T\$$.



# Finite State Machines

*From CS144*




- **Represent protocols using state machines**
  - Sender and receiver each have a state
  - Start in some initial state
  - Events cause each side to select a state

It's a generalization of DFAs!

- **Transition specifies action taken**
  - Specified as events/actions
  - E.g., software calls `send`/put packet on network
  - E.g., packet arrives/send acknowledgment

**From CS168**

Reducibility!

By definition, we need to output $y$ if and only if $y \in S$. That is, *answering membership queries reduces to solving the Heavy Hitters problem.* By the "membership problem," we mean the task of preprocessing a set $S$ to answer queries of the form "is $y \in S$"? (A hash table is the most common solution to this problem.) It is intuitive that you cannot correctly answer all membership queries for a set $S$ without storing $S$ (thereby using linear, rather than constant, space) — if you throw some of $S$ out, you might get a query asking about the part you threw out, and you won't know the answer. It's not too hard to make this idea precise using the Pigeonhole Principle.[5]

A Myhill-Nerode-style argument!

**From CS154**

# Kolmogorov Complexity (1960's)

Definition: The *shortest description of* $x$, denoted as $d(x)$, is the lexicographically shortest string $\langle M,w \rangle$ such that $M(w)$ halts with only $x$ on its tape.

Definition: The *Kolmogorov complexity of* $x$, denoted as $K(x)$, is $|d(x)|$.

Using Turing machines to define intrinsic information content!

You’ve given yourself the foundation to tackle problems from all over computer science.

There’s so much more to explore.
Where should you go next?

# Course Recommendations

## *Theoryland*

- CS154 — *Complexity*
- Phil 151 — *Computability*
- Phil 152 — *Computability*
- Math 107 — *Graphs*
- Math 108 — *Graphs*
- Math 113 — *Functions*
- Math 120 — *Functions*
- Math 161 — *Set Theory*
- Math 152 — *Number Theory*

## *Applications*

- CS124 — *Languages / Automata*, *Graphs*
- CS143 — *Languages / Automata*
- CS161 — *Graphs*
- CS224W — *Functions*
- CS242 — *Functions*
- CS243 — *Graphs*
- CS246 — *Functions*
- CS251 — *Functions*
- CS255 — *Functions*

# Final Thoughts

***There are more problems to solve than there are programs capable of solving them.***

There is so much more to explore and so many big questions to ask – ***many of which haven't been asked yet!***

You now know what problems we can solve, what problems we can't solve, and what problems we believe we can't solve efficiently.


Theory
Practice

# A Huge Round of Thanks!