CS103
Fall 2025

Lecture 26:

Complexity Theory

Part 2 of 2

Recap from Last Time

The Complexity Class **P**

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

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

- Intuitively, **P** contains all decision problems that can be solved efficiently.
- This is like class **R**, except with "efficiently" tacked onto the end.

The Complexity Class **NP**

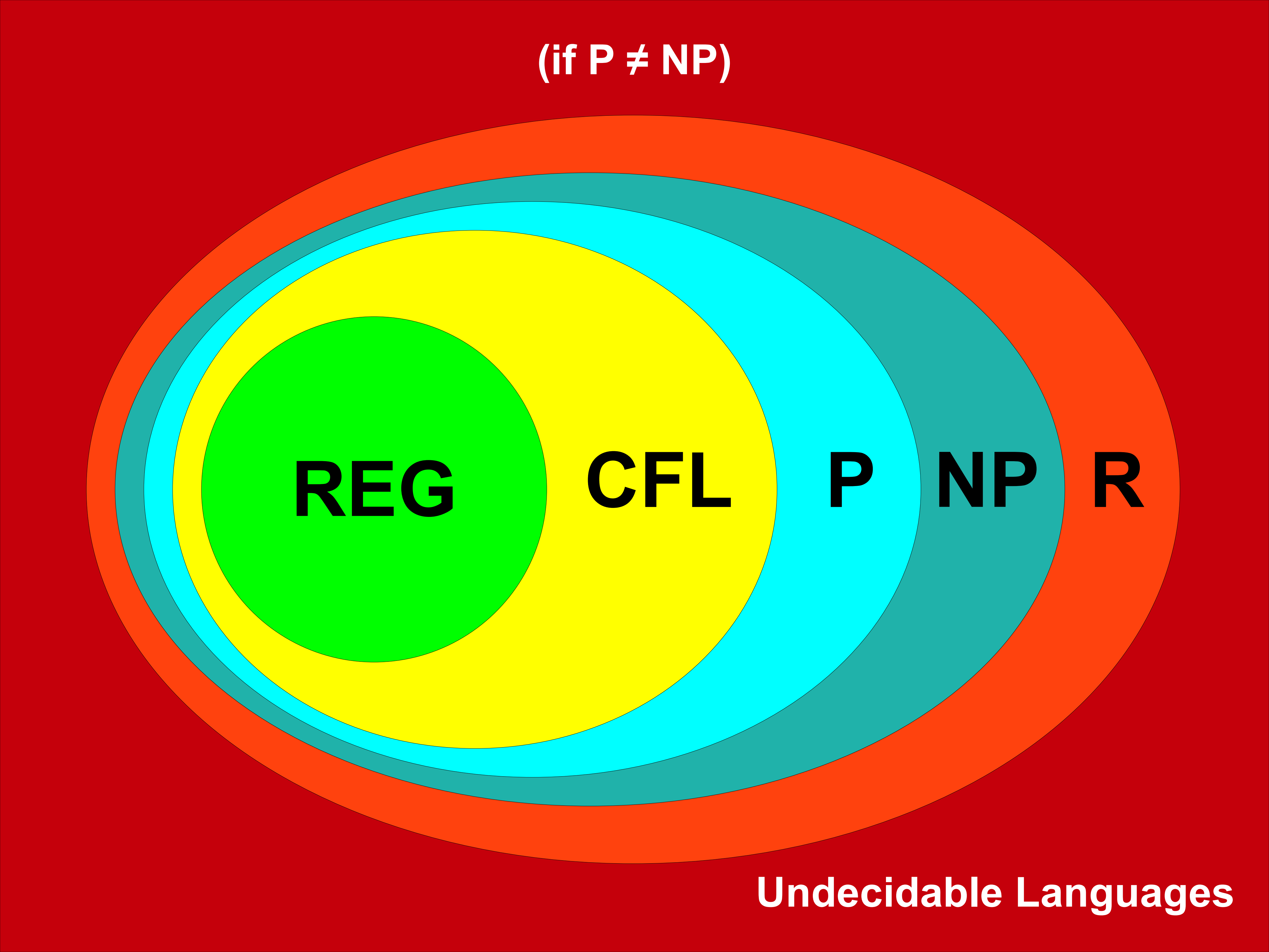
- The complexity class **NP** (***nondeterministic polynomial time***) contains all problems that can be verified in polynomial time.
- Formally:

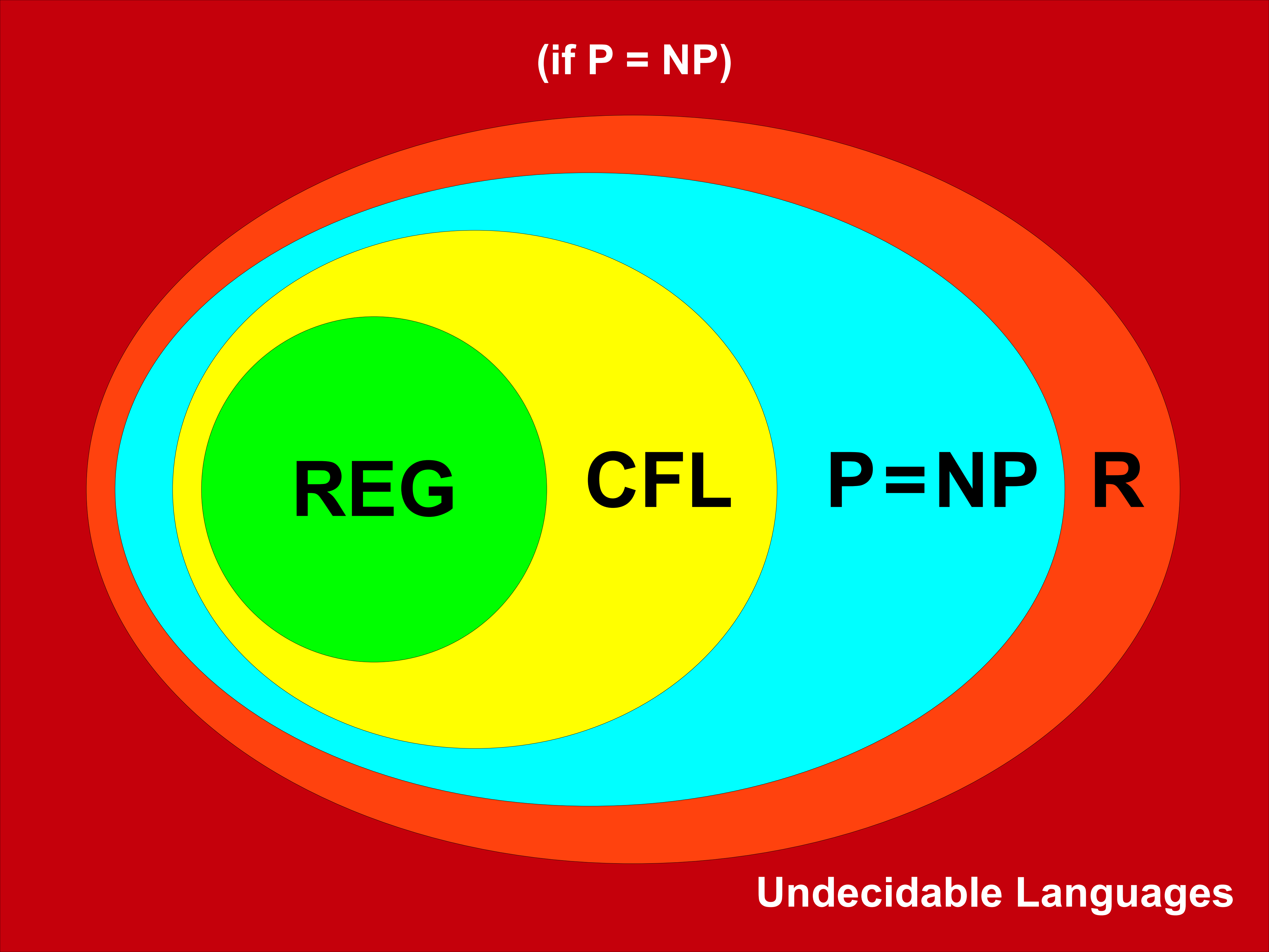
  NP = { L | There is a polynomial-time verifier for L }

- Intuitively, **NP** is the set of problems where "yes" answers can be checked efficiently.
- This is like the class **RE**, but with "efficiently" tacked on to the definition.

The Biggest Unsolved Problem in Theoretical Computer Science:

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


(if P ≠ NP)
REG
CFL
P
NP
R
Undecidable Languages


(if P = NP)
REG
CFL
P = NP
R
Undecidable Languages

Adapting Our Techniques

- We already know **R** ≠ **RE**. So does that mean **P** ≠ **NP**?
- 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**?

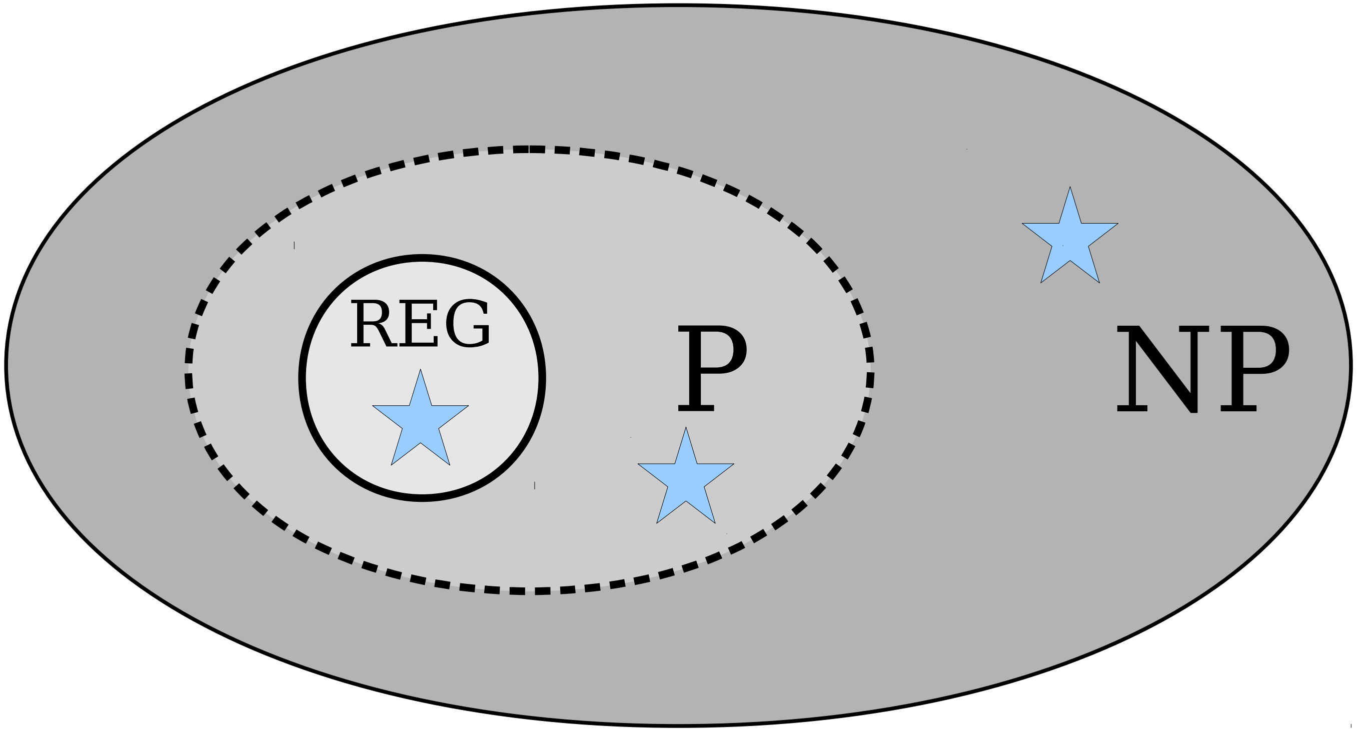
Theorem (Baker-Gill-Solovay): Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Proof: Take CS154!

So how *are* we going to reason about **P** and **NP**?

New Stuff!

An Initial Observation



Problems in **NP** vary widely in their difficulty, even if **P** = **NP**.

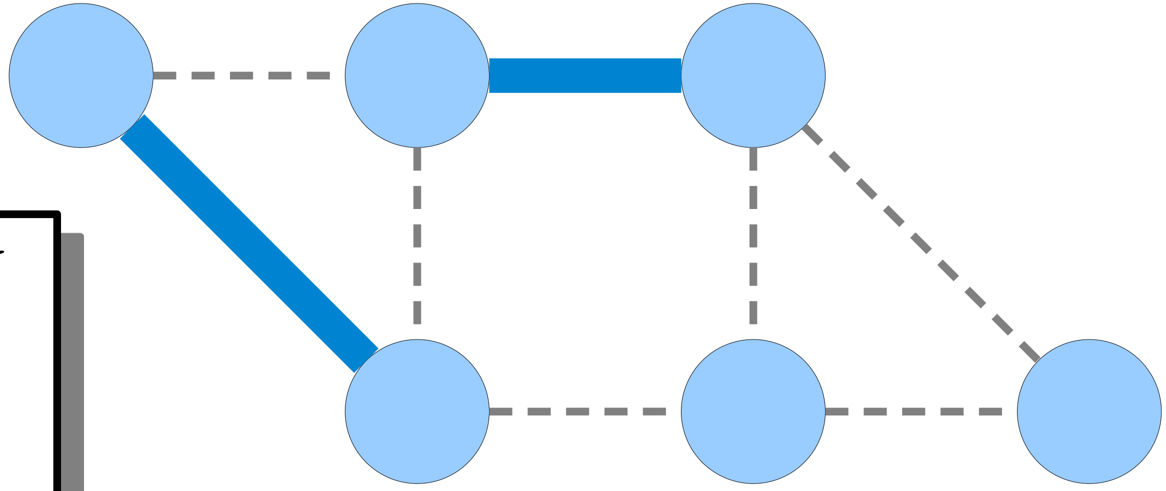
How can we rank the relative difficulties of problems?

Reducibility

Maximum Matching

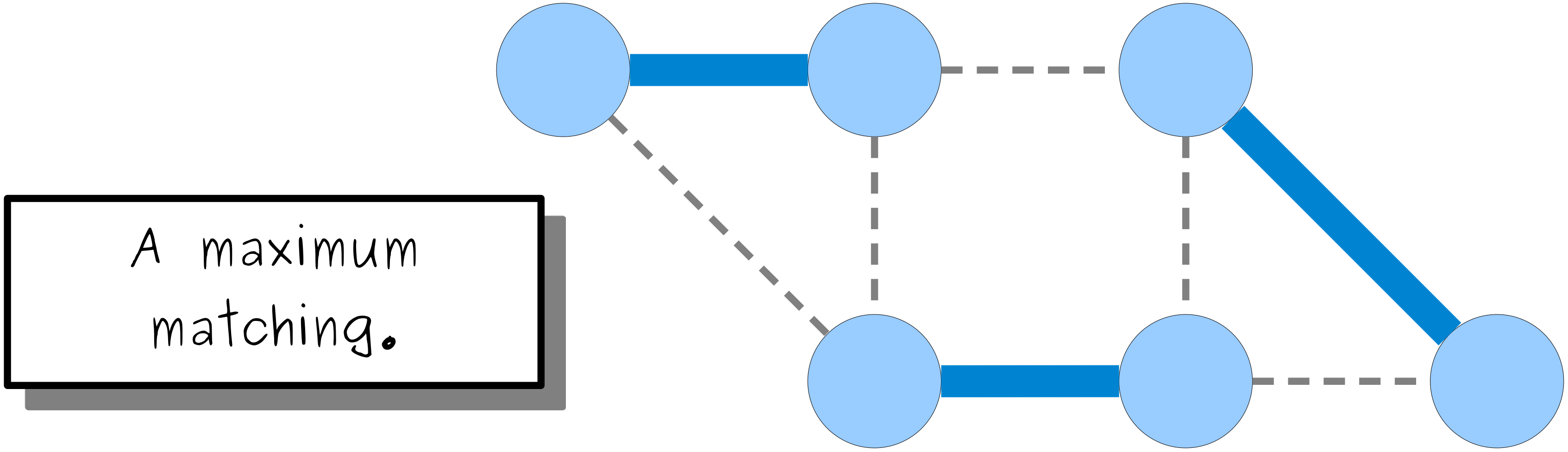
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.

A matching, but not a maximum matching.



Maximum Matching

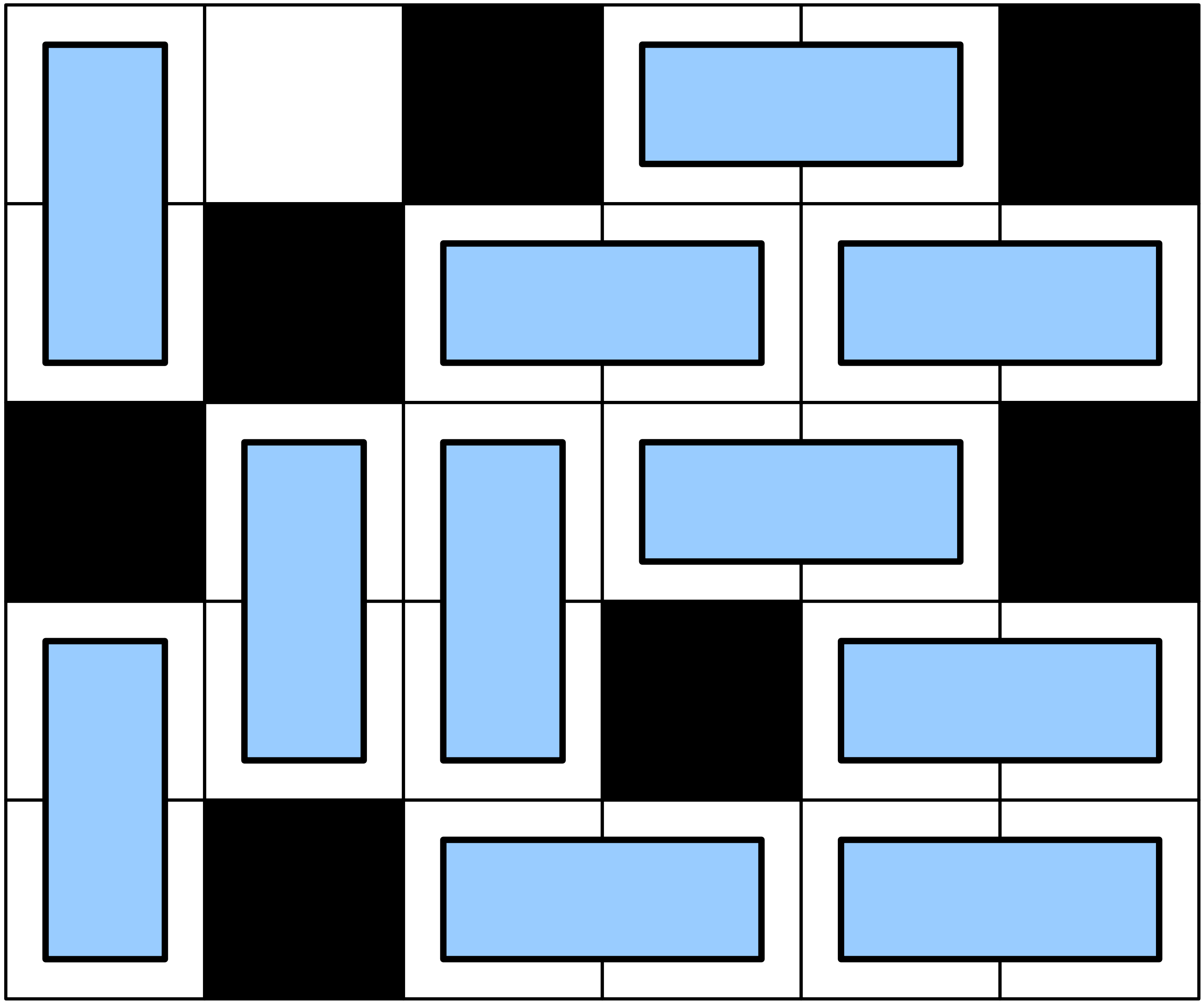
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



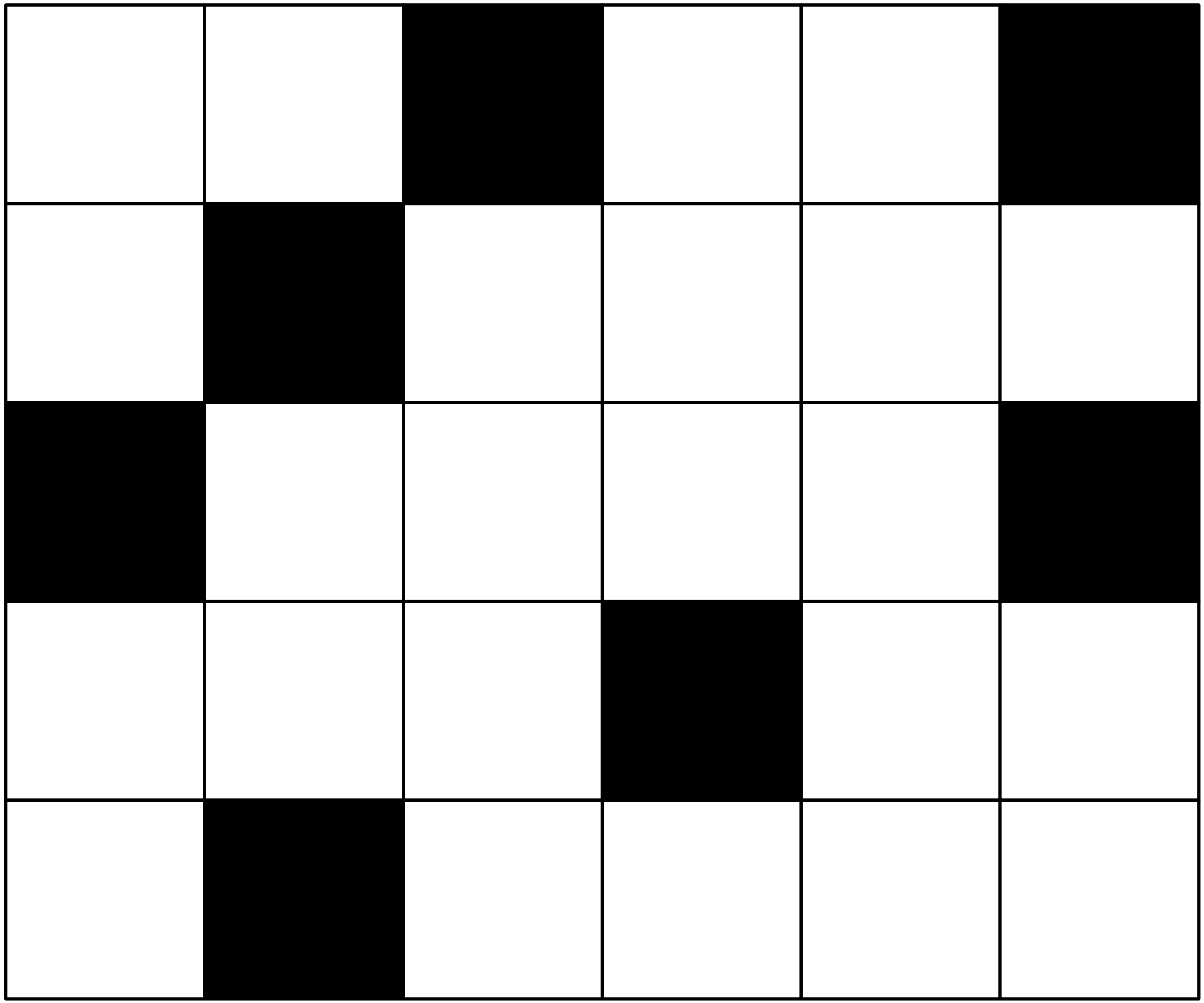
Maximum Matching

- Jack Edmonds' paper "Paths, Trees, and Flowers" describes a polynomial-time algorithm for finding maximum matchings.
  - He's the guy from last time with the quote about "better than decidable."
  - He's also the Edmonds in "Cobham-Edmonds Thesis."
- Using this fact, what other problems can we solve?

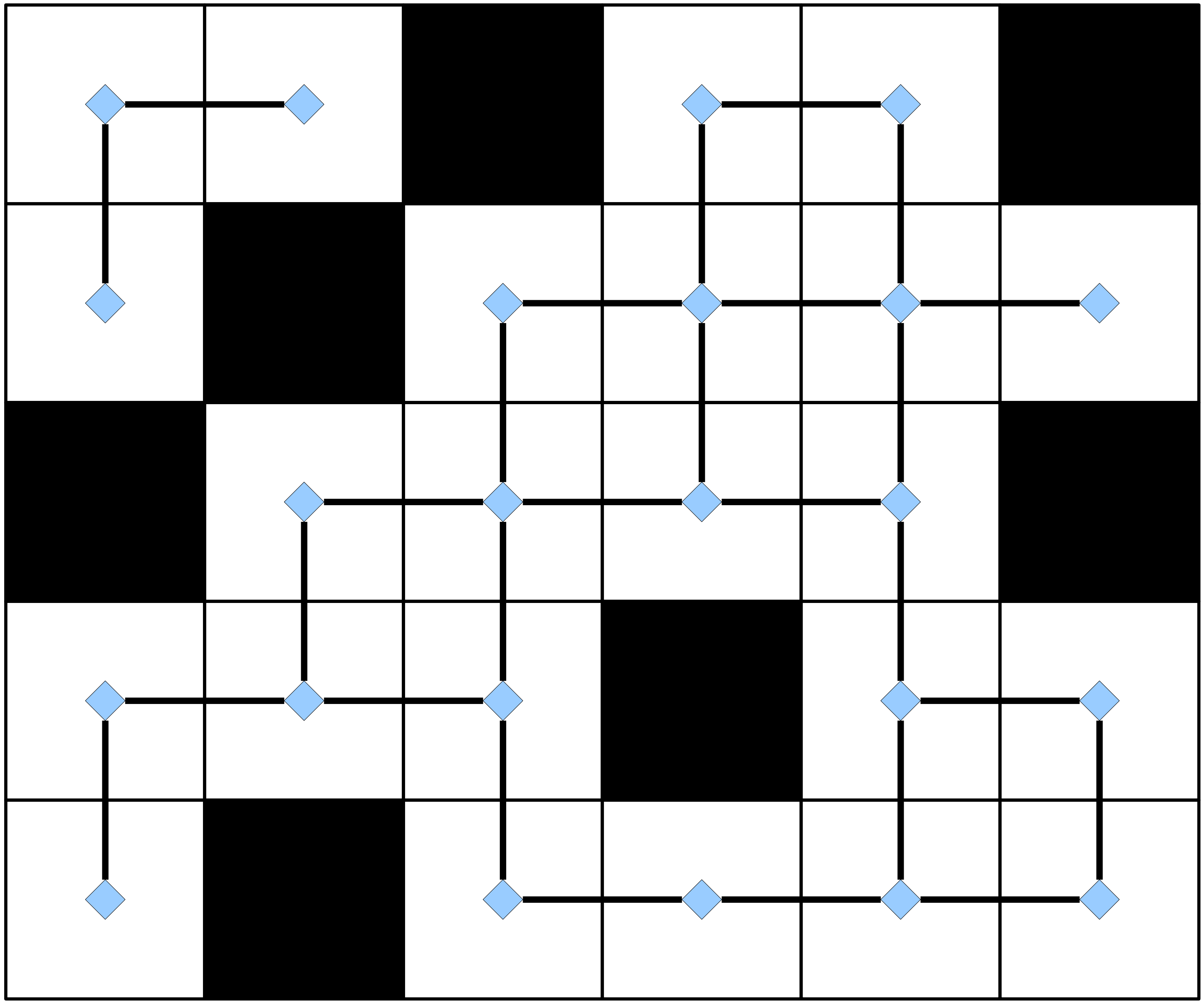
Domino Tiling



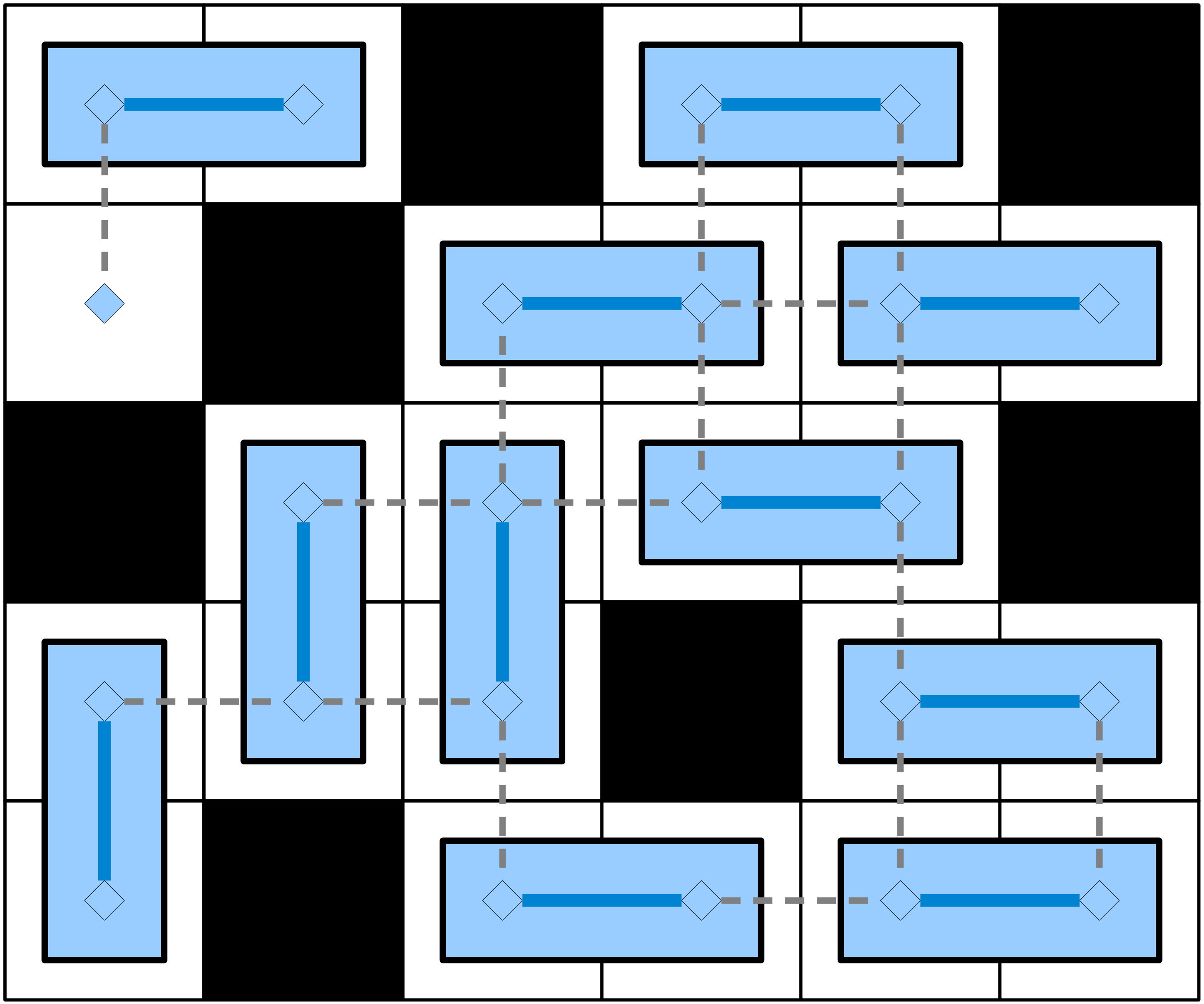
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



```
bool canPlaceDominoes(Grid G, int k) {
  return hasMatching(gridToGraph(G), k);
}
```

Which of the following is the most reasonable conclusion to draw, given the existence of the above function?

A. Solving domino tiling on a 2D grid can't be "harder" than solving maximum matching.

B. Solving maximum matching can't be "harder" than solving domino tiling on a 2D grid.

C. Both A and B.

Answer at ***https://cs103.stanford.edu/pollev***

Intuition:

Tiling a grid with dominoes can't be "harder" than solving maximum matching, because if we can solve maximum matching efficiently, we can solve domino tiling efficiently.

Another Example

Satisfiability

- A propositional logic formula φ is called ***satisfiable*** if there is some assignment to its variables that makes it evaluate to true.
- Which of the following formulas are satisfiable?

$$p \land q$$

$$p \land \neg p$$

$$p \to (q \land \neg q)$$

- An assignment of true and false to the variables of φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

- The ***boolean satisfiability problem*** (***SAT***) is the following:

Given a propositional logic formula φ, is φ satisfiable?

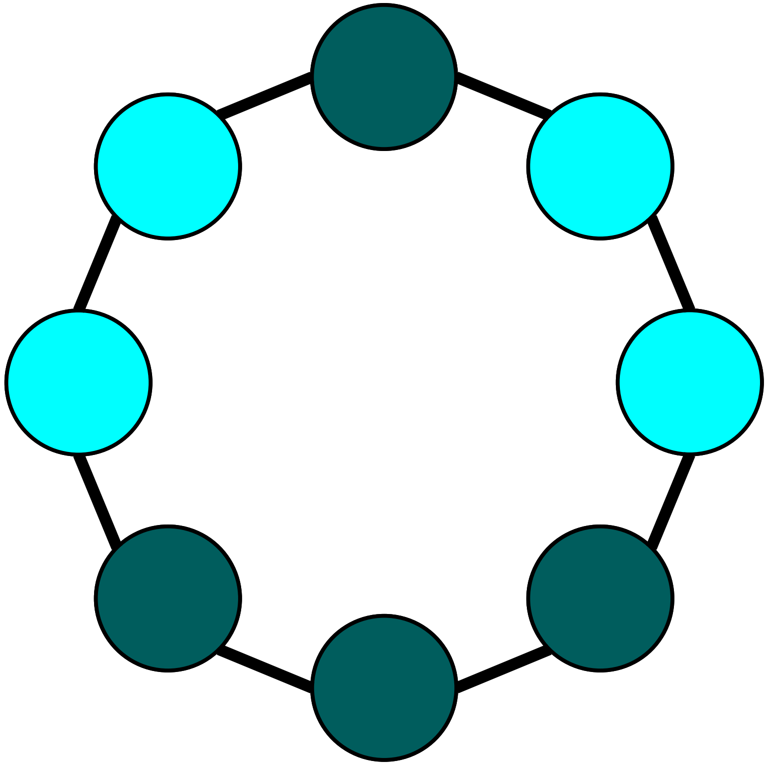
- Formally:

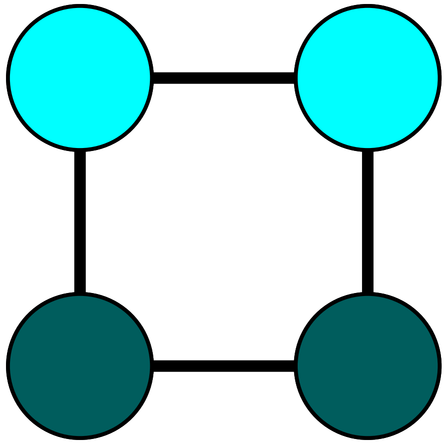
$$\textbf{\textit{SAT}} = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable PL formula} \}$$

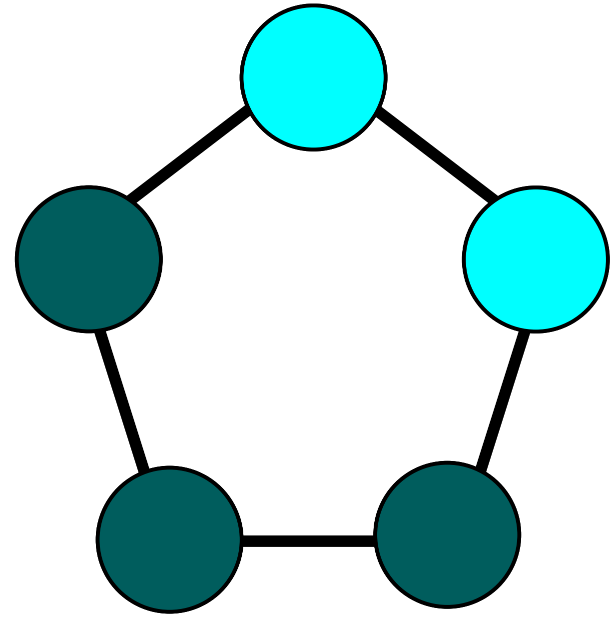
- Finding good algorithms for SAT is an active area of research for reasons we'll discuss later today.
- We have some pretty decent algorithms for solving SAT reasonably quickly most of the time.
- Given this, what other problems can we solve?

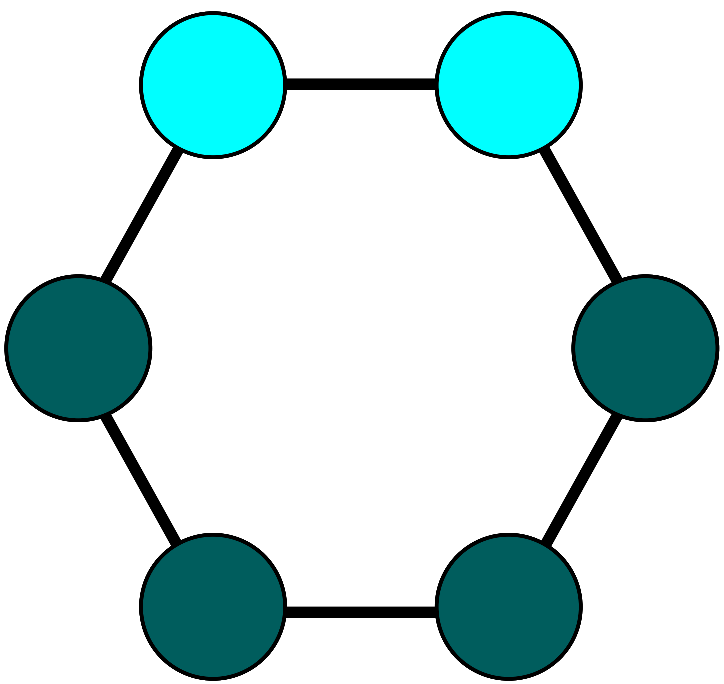
Lights Out

- You're given a ring of pushbuttons. Each pushbutton has a light that is either ON or OFF.
- If you push a button, it toggles the state of the two adjacent lights in the ring. (Lights that are ON turn OFF and vice-versa.)
- ***Question:*** Given an initial configuration of lights, can you turn all the lights off?





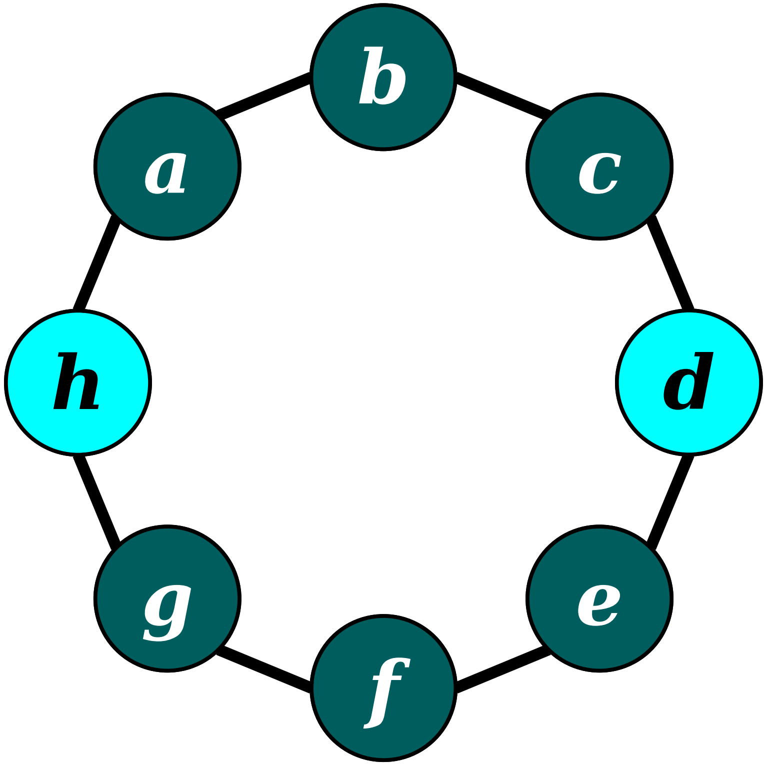




In which of these rings can you turn off all the lights?

Answer at ***https://cs103.stanford.edu/pollev***

Solving Lights-Out With a SAT Solver



$$
\begin{aligned}
&(h \leftrightarrow b) \land \\
&(a \leftrightarrow c) \land \\
&(b \leftrightarrow d) \land \\
\neg&(c \leftrightarrow e) \land \\
&(d \leftrightarrow f) \land \\
&(e \leftrightarrow g) \land \\
&(f \leftrightarrow h) \land \\
\neg&(a \leftrightarrow g)
\end{aligned}
$$

Observation 1: We never need to press the same button twice.

Observation 2: Button press order doesn't matter.

Idea: Our propositional formula will have one variable per button, indicating whether we press it.

Observation 3: A light that is initially off stays off when an even number of adjacent lights are pressed.

Observation 4: A light that is initially on ends off when an odd number of adjacent lights are pressed.

In Pseudocode

```
bool canTurnLightsOff(LightRing r) {
  return isSatisfiable(ringToFormula(r));
}
```

Intuition:

Solving Lights Out can’t be “harder” than solving SAT because if we can solve SAT efficiently, we can solve Lights Out efficiently.

```
bool canPlaceDominoes(Grid G, int k) {
  return hasMatching(gridToGraph(G), k);
}

bool canTurnLightsOff(LightRing r) {
  return isSatisfiable(ringToFormula(r));
}
```

```
bool solveProblemA(string input) {
    return solveProblemB(translate(input));
}
```

Intuition:

Problem *A* can't be “harder” than problem *B*, because solving problem *B* lets us solve problem *A*.

```
bool solveProblemA(string input) {
    return solveProblemB(translate(input));
}
```

- If *A* and *B* are problems where it's possible to solve problem *A* using the strategy shown above*, we write

$$A \leq_{\mathrm{p}} B.$$

- We say that ***A is polynomial-time reducible to B***.

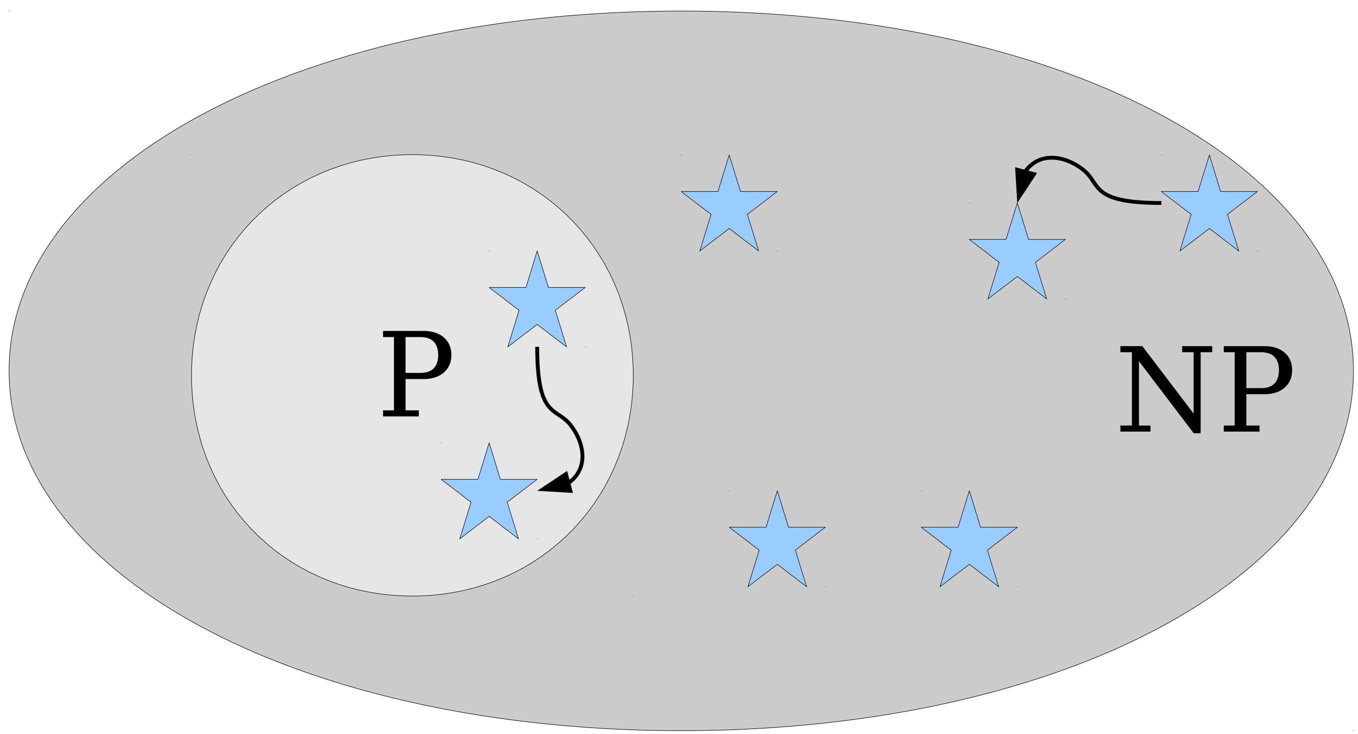
* Assuming that `translate` runs in polynomial time.

```
bool solveProblemA(string input) {
    return solveProblemB(translate(input));
}
```

- This is a powerful general problem-solving technique. You'll see it a lot in CS161.

Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



This $\leq_p$ relation lets us rank the relative difficulties of problems in **P** and **NP**.

What else can we do with it?

Time-Out for Announcements!

Please evaluate this course on Axess.

Your feedback makes a difference.

Join the CS Belonging & Engagement Committee for our

CODA STUDY TAKEOVER

for prospective and current CS students

Finals lock-in. Your friends, free Chick-fil-A, and coffee + tea.

Wednesday, December 3rd 5-8p.m.
CODA Basement

SEE YOU THERE!

RSVP HERE

https://partiful.com/e/U8C6MPFqaGNuTLDE42iM

Don Knuth Lecture

- Don Knuth, a living legend in CS, is giving his 29th annual Christmas Lecture tomorrow (***Thursday, December 4th***) at ***6PM*** in ***Nvidia Auditorium***.
- The topic is "Adventures with Knight's Tours," which I expect will involve some really neat tricks with graph theory.
- Highly recommended!

Final Exam Logistics

- Our final exam is ***Wednesday, December 10th*** from ***3:30 – 6:30 PM***.
  - Seating assignments will be online soon; we'll make an announcement when they're ready.
  - Expect seating assignments to change from the first two exams.
- The final exam is cumulative, covering topics from PS0 – PS9 and L00 – L26. The format is similar to that of the midterms, with a mix of short-answer questions and formal written proofs.
- Like the midterms, it's closed-book, closed-computer, and limited-note. You can bring one double-sided 8.5" × 11" notes sheet with you.
- Students needing alternate exam times: you should have heard from us already with your exam time/location. Contact us ASAP if you haven't.

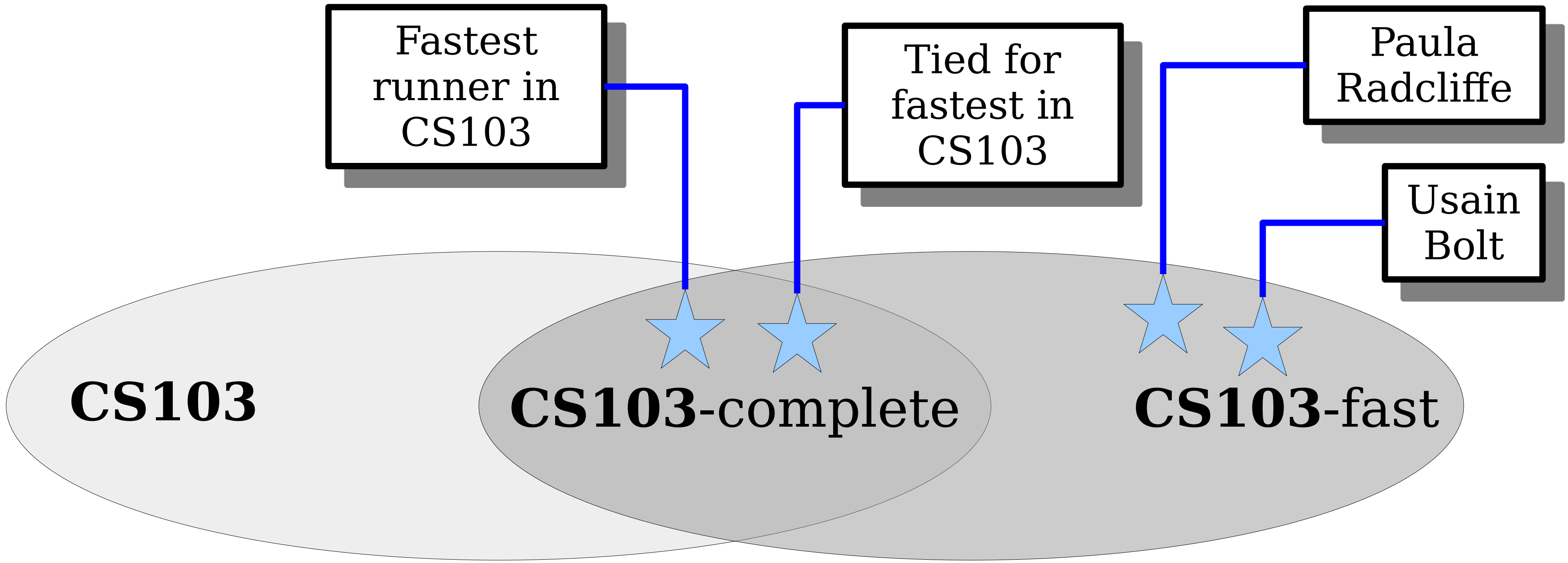
Preparing for the Exam

- Kaia will be holding a review session ***Friday*** from ***4:30PM – 5:30PM*** in ***Thornton 102***.
  - This review session will be recorded, but we highly recommend attending in person. You'll get way more out of it if you do!
- We've also released the Cumulative Practice Problems list, a gigantic searchable database of problems you can use to brush up on whatever topics you need the most practice with.
- As always, ***keep the TAs in the loop when studying***! That's what we're here for.

Back to CS103!

NP-Hardness and **NP**-Completeness

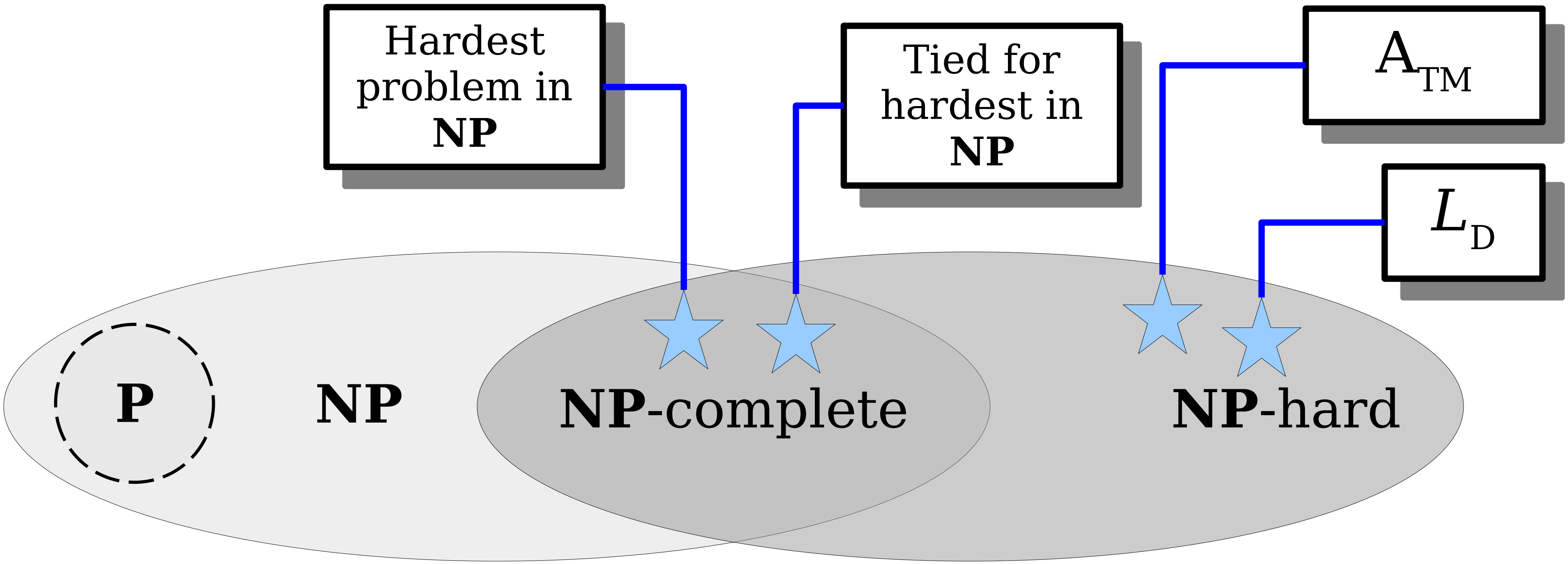
An Analogy: Running Really Fast



For people A and B, we say $\boldsymbol{A \leq_r B}$ if
A's top running speed is at most B's top speed.
(Intuitively: B can run at least as fast as A.)

We say that person P is ***CS103-fast*** if
$\forall A \in \textbf{CS103}.\ A \leq_r P.$
(How fast are you if you're CS103-fast?)

We say that person P is ***CS103-complete*** if
$P \in$ **CS103** and P is **CS103**-fast.
(How fast are you if you're CS103-complete?)



For languages A and B, we say $\boldsymbol{A \leq_p B}$ if A reduces to B in polynomial time.

(Intuitively: B is at least as hard as A.)

We say that a language L is ***NP-hard*** if

$\forall A \in \mathbf{NP}.\ A \leq_p L.$

(How hard is a problem that's NP-hard?)

We say that a language L is ***NP-complete*** if

$L \in \mathbf{NP}$ and L is $\mathbf{NP}$-hard.

(How hard is a problem that's NP-complete?)

Intuition: The **NP**-complete problems are the hardest problems in **NP**.

If we can determine how hard those problems are, it would tell us a lot about the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question.

The Tantalizing Truth

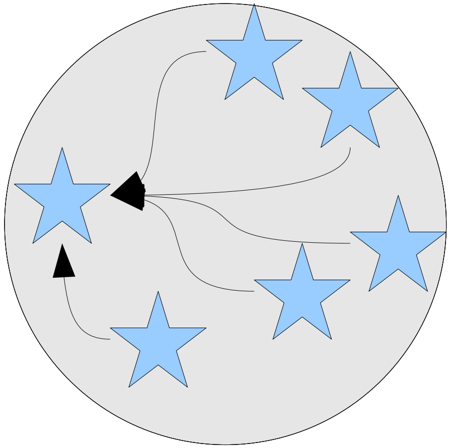
Theorem: If *any* **NP**-complete language is in **P**, then **P** = **NP**.

Intuition: This means the hardest problems in **NP** aren't actually that hard. We can solve them in polynomial time. So that means we can solve all problems in **NP** in polynomial time.

The Tantalizing Truth

Theorem: If *any* **NP**-complete language is in **P**, then **P** = **NP**.

Proof: Suppose that L is **NP**-complete and $L \in$ **P**. Now consider any arbitrary **NP** problem A. Since L is **NP**-complete, we know that $A \leq_p L$. Since $L \in$ **P** and $A \leq_p L$, we see that $A \in$ **P**. Since our choice of A was arbitrary, this means that **NP** ⊆ **P**, so **P** = **NP**. ■

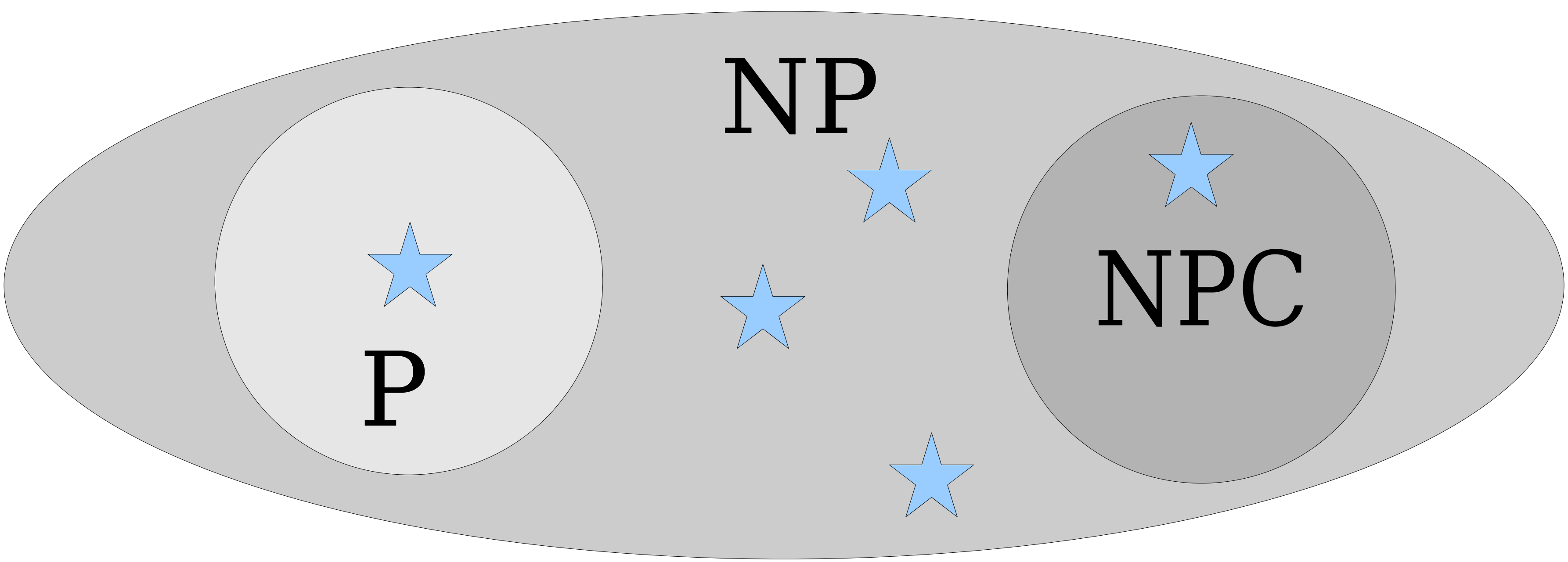


P = NP

The Tantalizing Truth

Theorem: If *any* **NP**-complete language is not in **P**, then **P** ≠ **NP**.

Proof: Suppose that L is an **NP**-complete language not in **P**. Since L is **NP**-complete, we know that $L \in$ **NP**. Therefore, we know that $L \in$ **NP** and $L \notin$ **P**, so **P** ≠ **NP**. ■



How do we even know NP-complete problems exist in the first place?

Theorem (Cook-Levin): SAT is **NP**-complete.

Proof Idea: To see that SAT ∈ **NP**, show how to make a polynomial-time verifier for it. Key idea: the certificate is a candidate satisfying assignment.

To show that SAT is **NP**-hard, given a polymomial-time verifier *V* for an arbitrary **NP** language *L*, for any string *w* you can construct a polynomially-sized formula φ(*w*) that says "there's a certificate *c* where *V* accepts ⟨*w*, *c*⟩." This formula is satisfiable if and only if *w* ∈ *L*, so deciding whether the formula is satisfiable decides whether *w* is in *L*. ■-*ish*

Proof: Take CS154!

Why All This Matters

- Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is equivalent to just figuring out how hard SAT is.

$$\text{SAT} \in \mathbf{P} \quad \leftrightarrow \quad \mathbf{P} = \mathbf{NP}$$

- We've turned a huge, abstract, theoretical problem about solving problems versus checking solutions into the concrete task of seeing how hard one problem is.
- You can get a sense for how little we know about algorithms and computation given that we can't yet answer this question!

Sample **NP**-Hard Problems

- ***Computational biology:*** Given a set of genomes, what is the most probable evolutionary tree that would give rise to those genomes? *(Maximum parsimony problem)*
- ***Game theory:*** Given an arbitrary perfect-information, finite, two-player game, who wins? *(Generalized geography problem)*
- ***Operations research:*** Given a set of jobs and workers who can perform those tasks in parallel, can you complete all the jobs within some time bound? *(Job scheduling problem)*
- ***Machine learning:*** Given a set of data, find the simplest way of modeling the statistical patterns in that data. *(Bayesian network inference problem)*
- ***Medicine:*** Given a group of people who need kidneys and a group of kidney donors, find the maximum number of people who can receive transplants. *(Cycle cover problem)*
- ***Systems:*** Given a set of processes and a number of processors, find the optimal way to assign those tasks so that they complete as soon as possible. *(Processor scheduling problem)*

Why All This Matters

- You will almost certainly encounter **NP**-hard problems in practice – they're everywhere!
- If a problem is **NP**-hard, then there is no known algorithm for that problem that
  - is efficient on all inputs,
  - always gives back the right answer, and
  - runs deterministically.
- ***Useful intuition:*** If you need to solve an **NP**-hard problem, you will either need to settle for an approximate answer, an answer that's likely but not necessarily right, or have to work on really small inputs.

Coda: What if $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is resolved?

Next Time

- ***Why All This Matters***
- ***Where to Go from Here***
- ***A Final "Your Questions"***
- ***Parting Words!***