# Complexity Theory

## Part Two

# Recap from Last Time

# The Complexity Class **P**

- The complexity class **P** (***polynomial time***) is defined as

  **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:

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

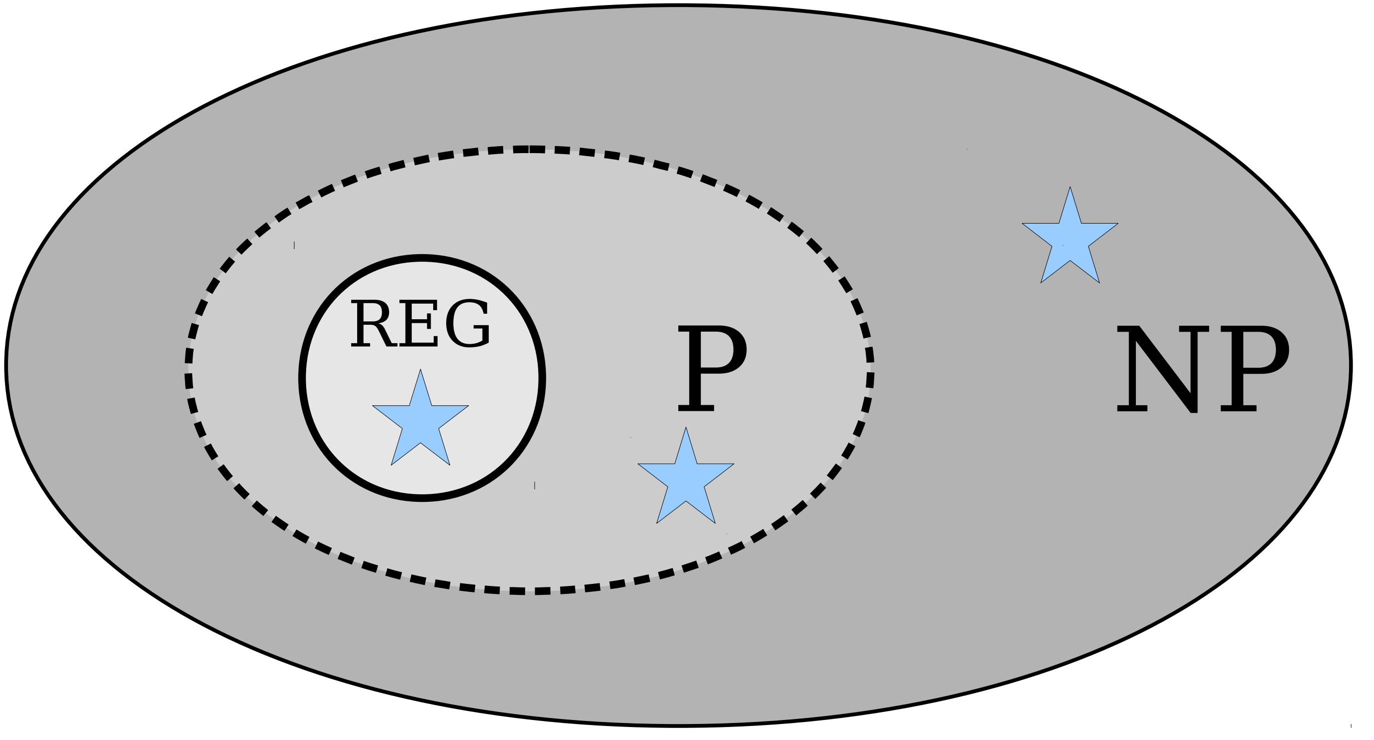
***Theorem (Baker-Gill-Solovay):*** Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

***Proof:*** Take CS154!

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

# New Stuff!

# A Challenge



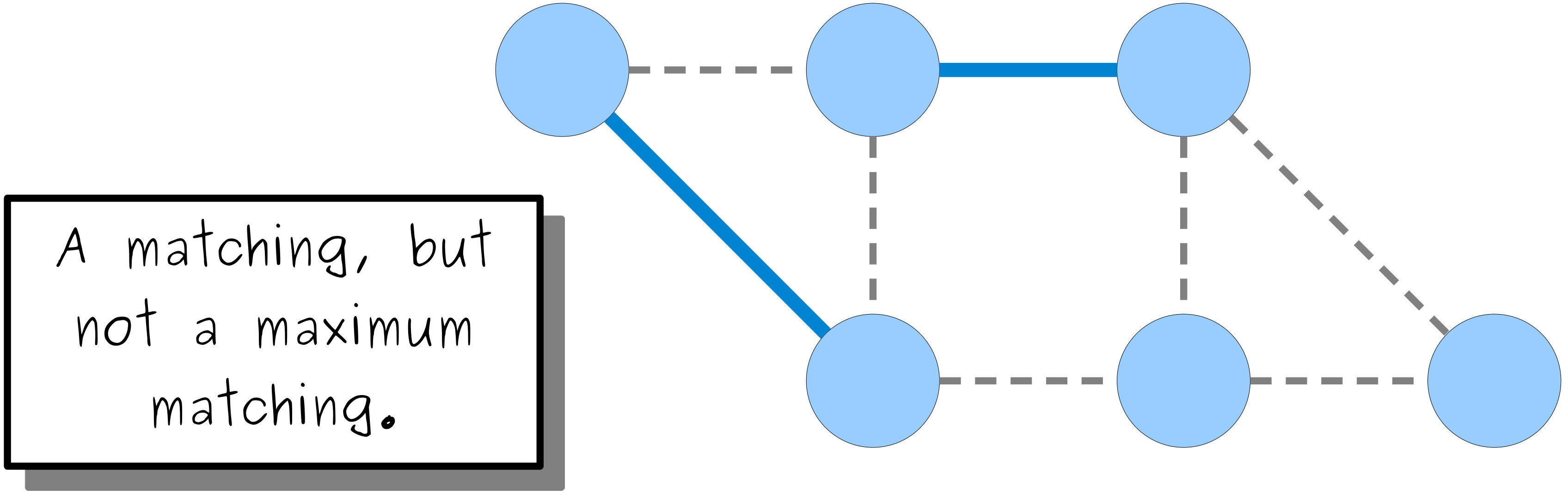

Problems in **NP** vary widely in their difficulty, even if **P** = **NP**.

How can we rank the relative difficulties of problems?

# Reducibility

# Maximum Matching

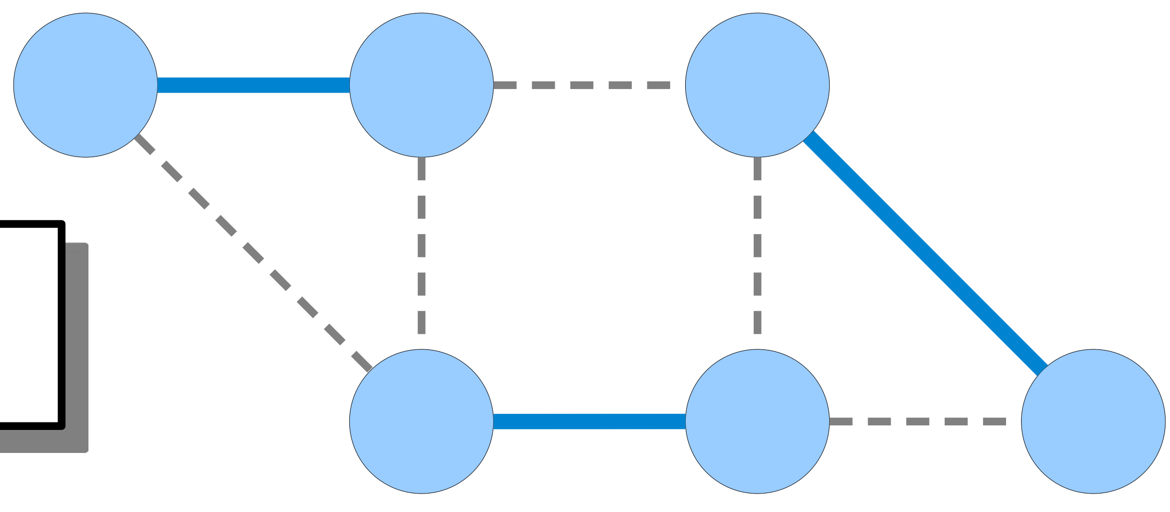
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



# Maximum Matching

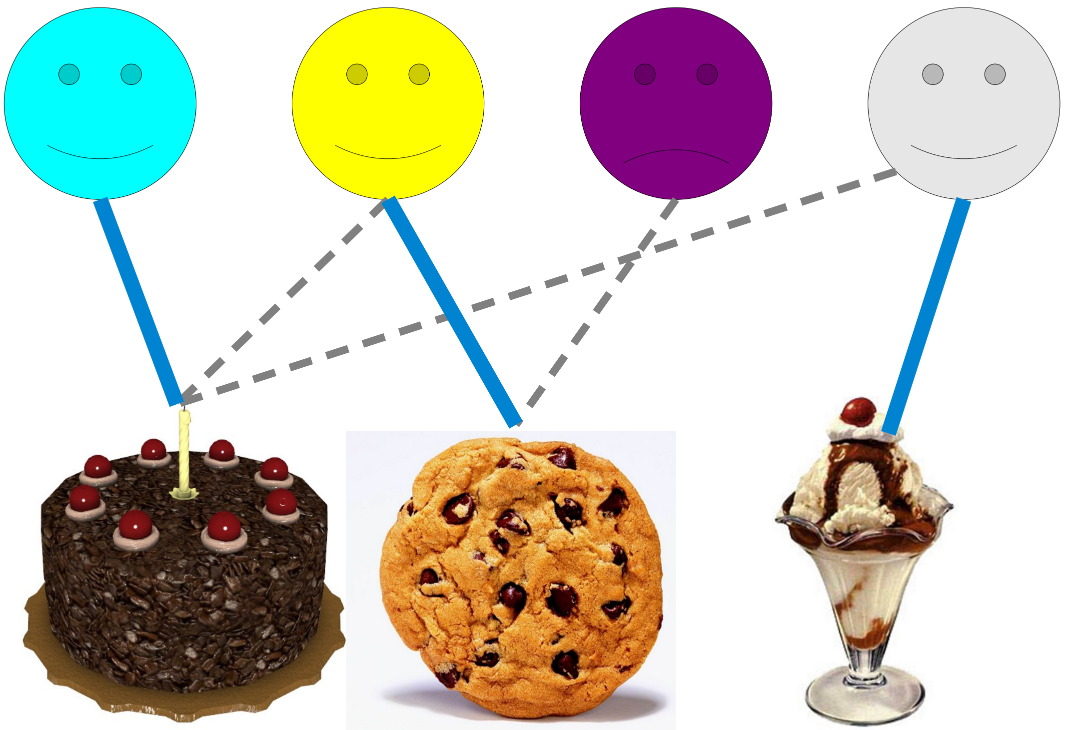
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.

A maximum matching.



# Maximum Matching

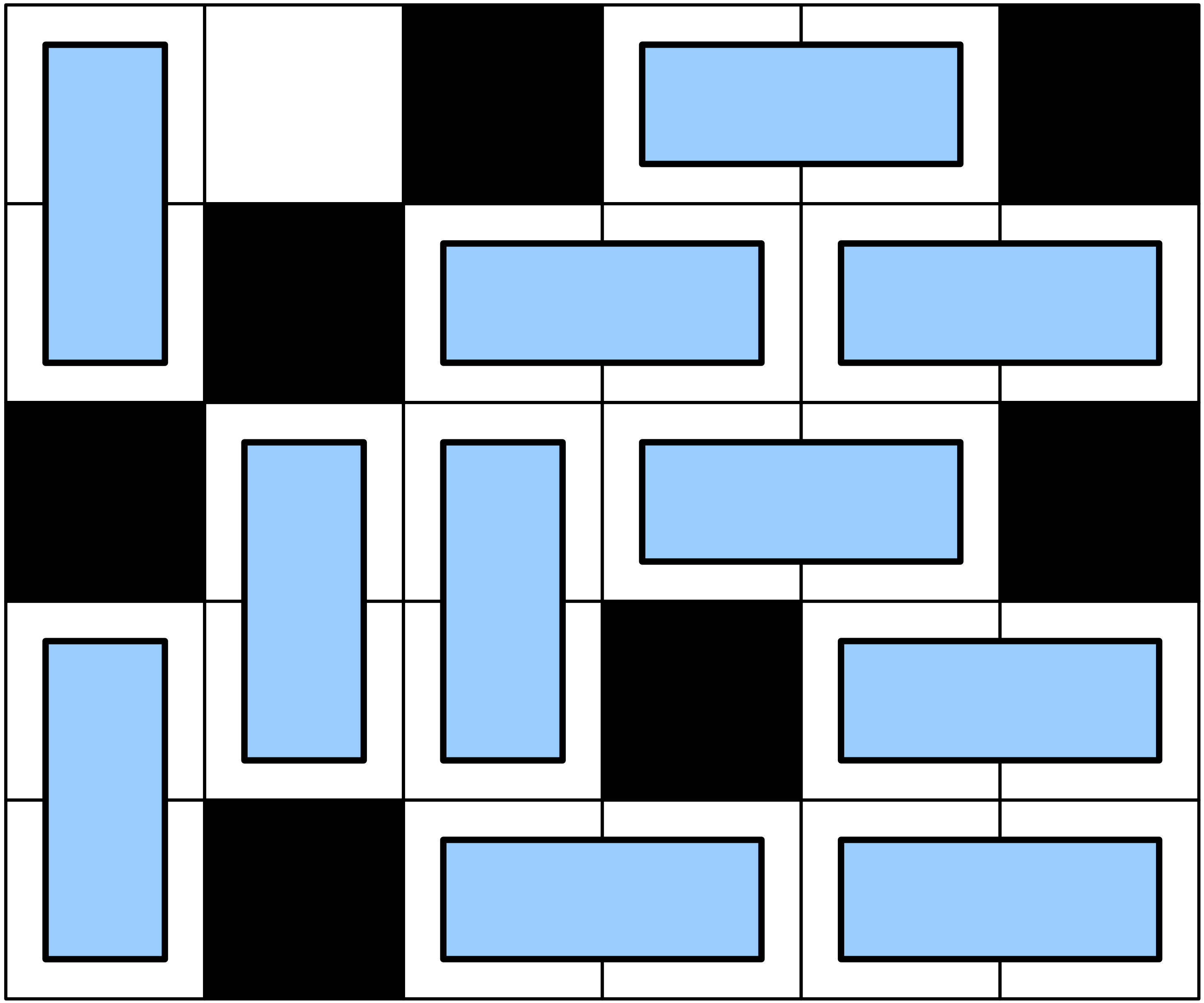
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



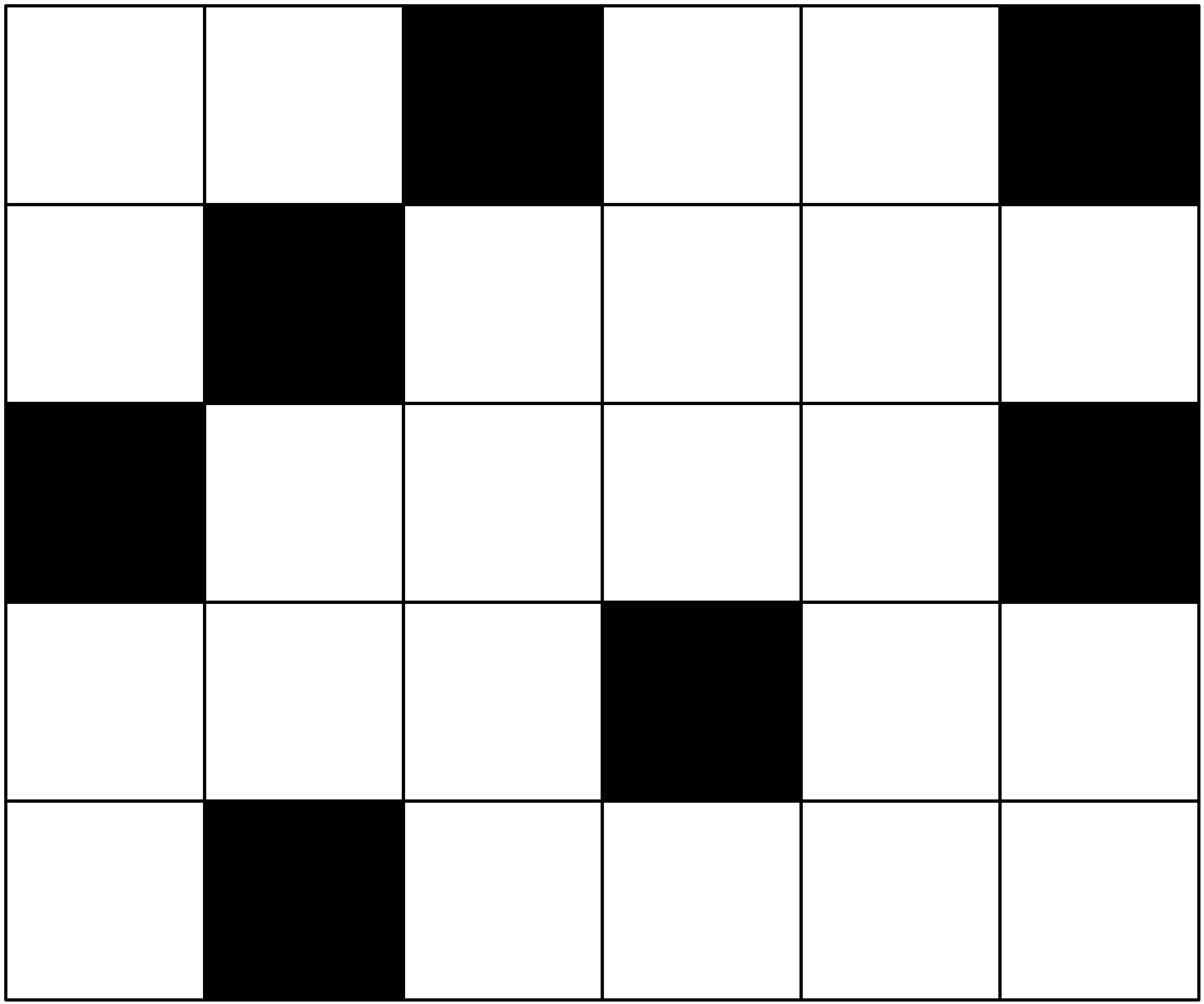
# Maximum Matching

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

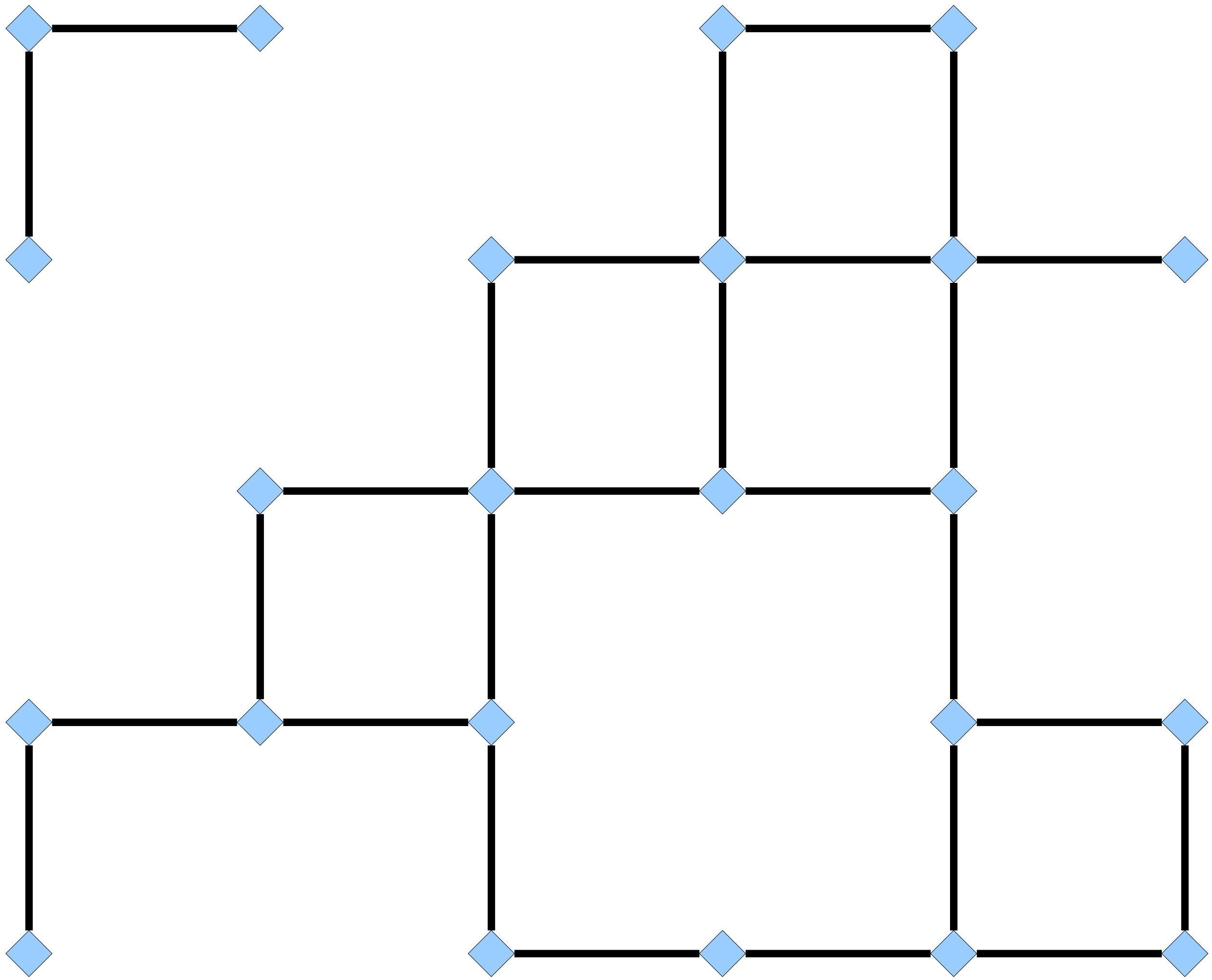
# Domino Tiling



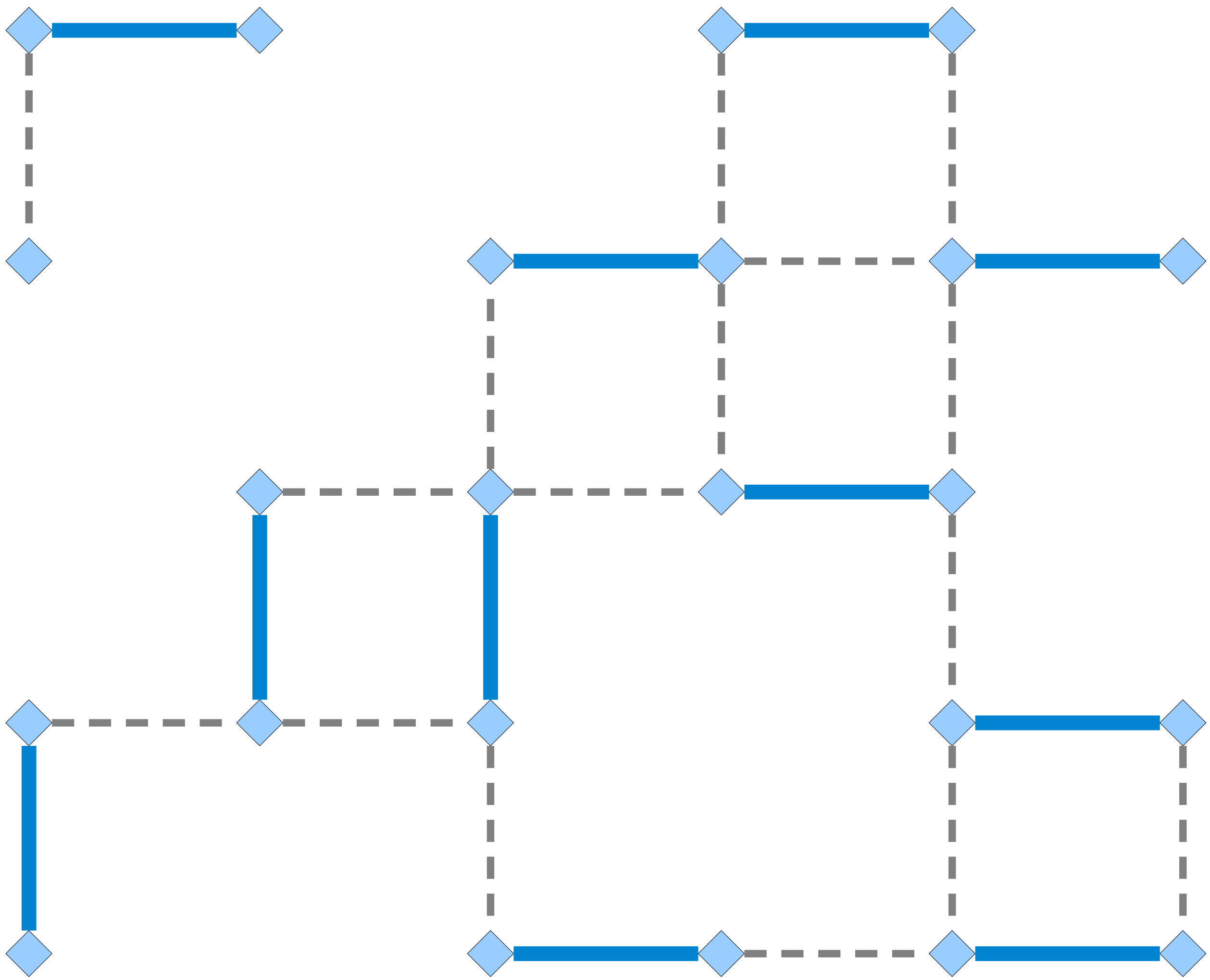
# Solving Domino Tiling



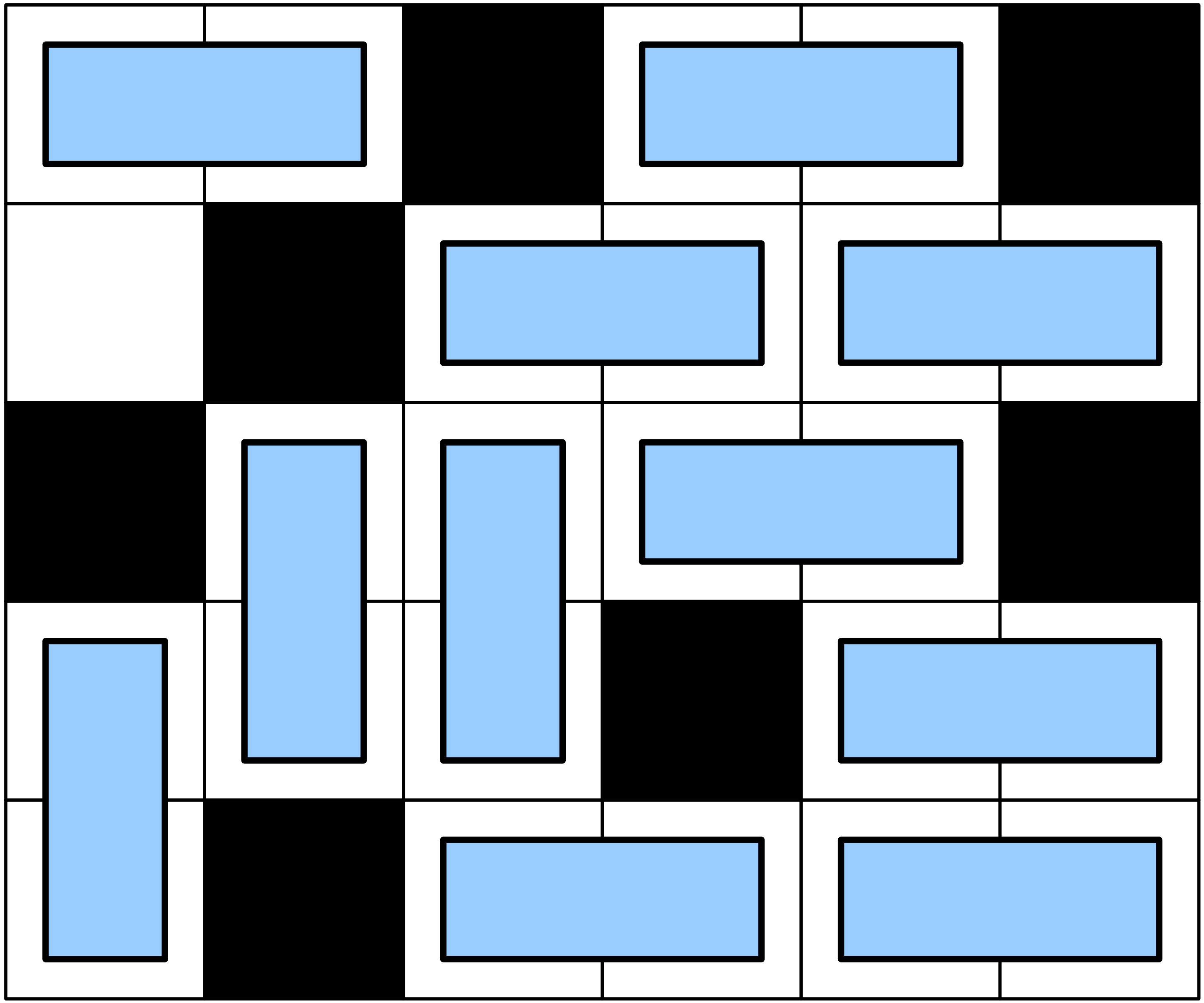
# Solving Domino Tiling



# Solving Domino Tiling



# Solving Domino Tiling



# In Pseudocode

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

***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

# Reachability

- Consider the following problem:

  **Given an directed graph *G* and nodes *s* and *t* in *G*, is there a path from *s* to *t*?**

- This problem can be solved in polynomial time (use DFS or BFS).

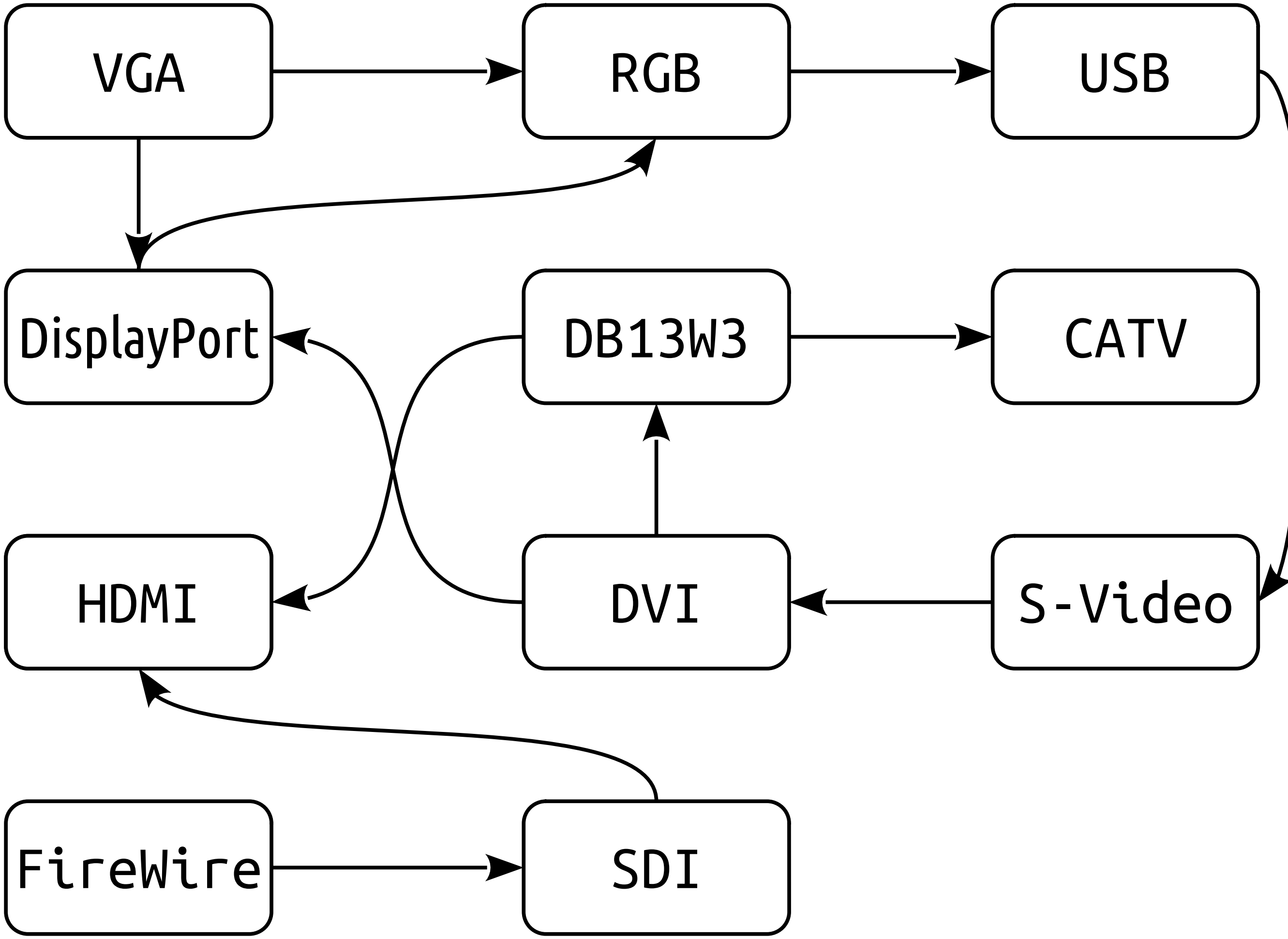
# Converter Conundrums

- Suppose that you want to plug your laptop into a projector.
- Your laptop only has a VGA output, but the projector needs HDMI input.
- You have a box of connectors that convert various types of input into various types of output (for example, VGA to DVI, DVI to DisplayPort, etc.)
- ***Question:*** Can you plug your laptop into the projector?

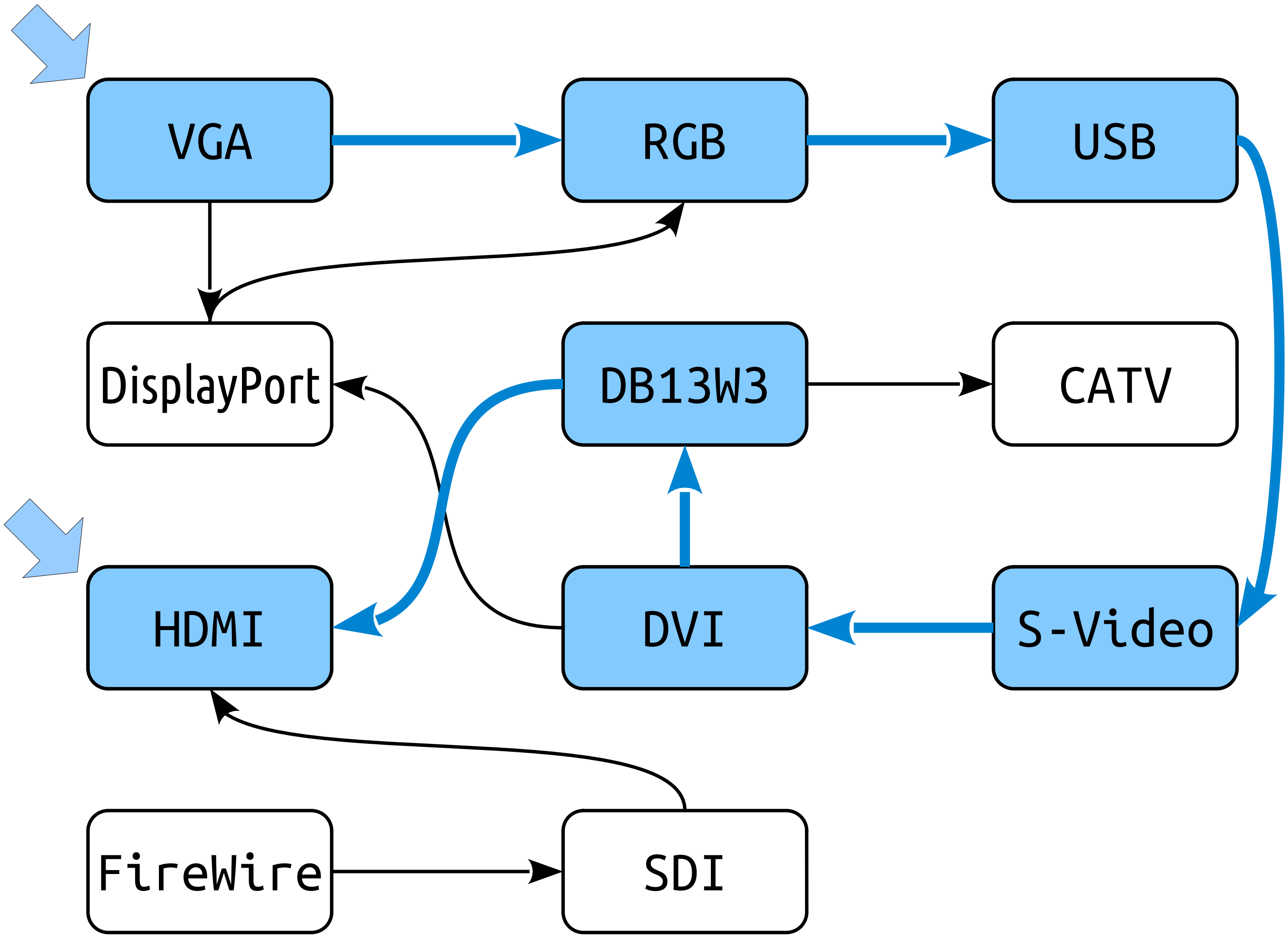
# Converter Conundrums

**Connectors**

RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



# Converter Conundrums



# In Pseudocode

```
bool canPlugIn(vector<Plug> plugs) {
  return isReachable(plugsToGraph(plugs),
                               VGA, HDMI);
}
```

***Intuition:***

Finding a way to plug a computer into a projector can't be “harder” than determining reachability in a graph, since if we can determine reachability in a graph, we can find a way to plug a computer into a projector.

```
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

$$A \leq_p B.$$

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

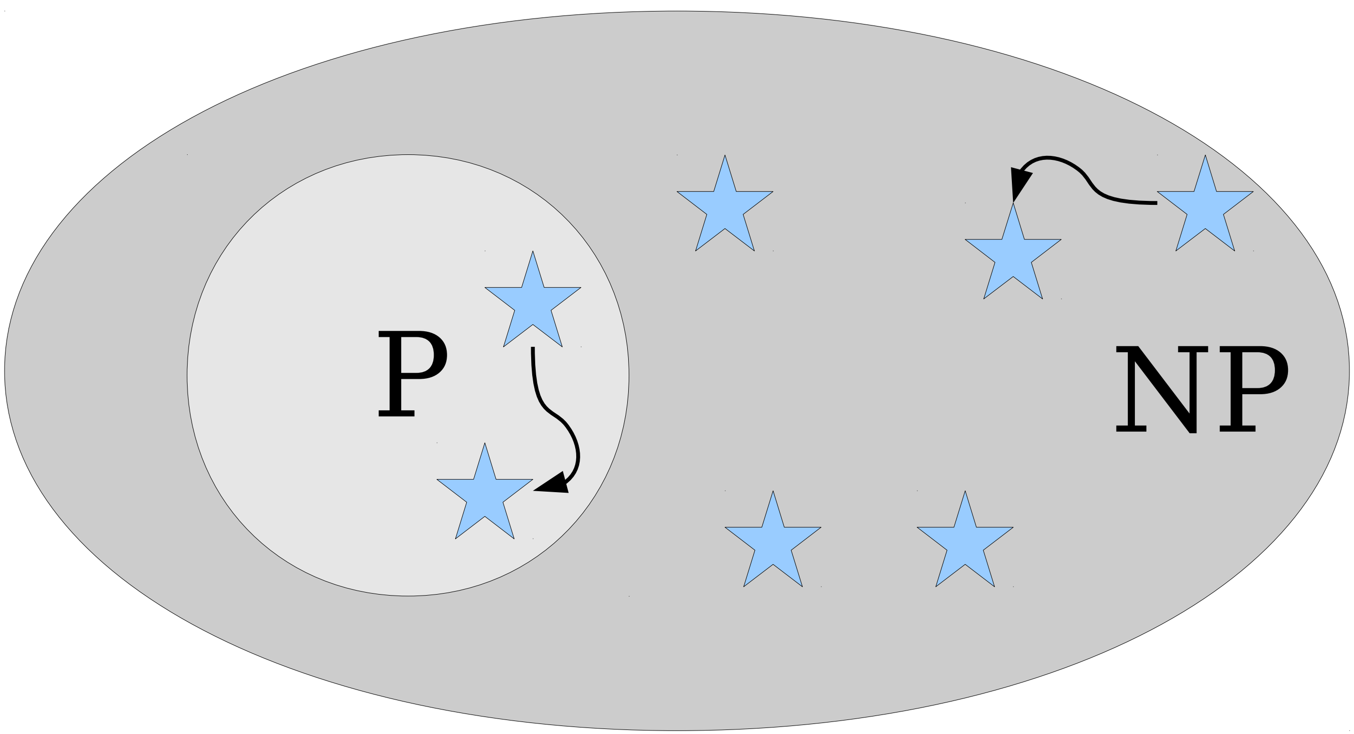
* Assuming that `translate` runs in polynomial time.

```
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.

# Final Exam Logistics

- Our final exam is on ***Tuesday, December 13^th^*** from ***8:30AM – 11:30AM***. It'll be held in ***Hewlett 200***.
- The final exam is cumulative and covers topics from PS1 – PS9 and L00 – L27. The format is similar to that of the midterm, 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 with OAE accommodations: we will be reaching out later this evening to coordinate alternate final exam times.

# Preparing for the Final Exam

- We’ve posted a gigantic compendium of CS103 practice problems on the course website.
- You can search for problems based on the topics they cover, whether solutions are available, whether they’re ones we particularly like, and whether they were used on past exams.
- As always, ***keep the TAs in the loop!*** Ask us questions if you have them, feel free to stop by office hours to discuss solutions, etc.

# Review Session

- Our fantastic TA Ophir will be holding a final exam review session this ***Sunday, December 11th*** starting at ***2:00PM*** in ***300-300***.
- Feel free to stop on by!
- SCPD students: we’ll have a Zoom link you can use to join, though it’ll be a little more low-tech than our standard lectures.

# Your Questions

“What do you think of this proof ChatGPT wrote for one of the practice problems posted (Bernoulli's Inequality w/ answer):”

PI

Bernoulli's inequality states that for any real number $r \geq -1$ and natural number n, we know that
$(1+r)^n \geq 1+rn$.
Prove Bernoulli's inequality by induction on n starting from 0.

Base Case: $n = 0$

In this case, the inequality states that $(1+r)^0 \geq 1+0r$.

*Since* $(1+r)^0 = 1$ *and* $1+0r = 1$, the inequality is satisfied.

Inductive Step: Assume that the inequality is true for some natural number k, i.e.

$(1+r)^k \geq 1+kr$.

We need to show that the inequality is also true for k+1, i.e. $(1+r)^{(k+1)} \geq 1+(k+1)r$.

We have that $(1+r)^{(k+1)} = (1+r)^k * (1+r) = (1+kr) * (1+r) = 1+kr + r+kr^2 = 1+(k+1)r + kr^2$.

Now, since $r \geq -1$, we have that $kr^2 \geq 0$, so $1+(k+1)r + kr^2 \geq 1+(k+1)r$.

Therefore, $(1+r)^{(k+1)} \geq 1+(k+1)r$, which completes the induction step.

Hence, by mathematical induction, the inequality is true for all natural numbers n.

# “whats the difference between cs 108 and CS111. RIP ”

CS111 gives an intro to systems software - multithreading, multiprocessing, file systems, etc. You get a lot of experience seeing how the operating system does its job and get really, really good at writing lower-level code in C++ and truly understanding how the machine executes instructions.

CS108 is all about object-oriented programming and explores how to build and test big, complicated programs. The focus is less on learning new technical concepts and more on good software engineering practice. The final project is a big, open-ended program done in a team, where you get to apply everything you've learned.

“whats a good rule of thumb for problems we cant solve? for instance, we are just hanging out doing some coding, trying to get the code to work, but actually we are being fooled because the code we are working on wont ever work ... how can we know? ”

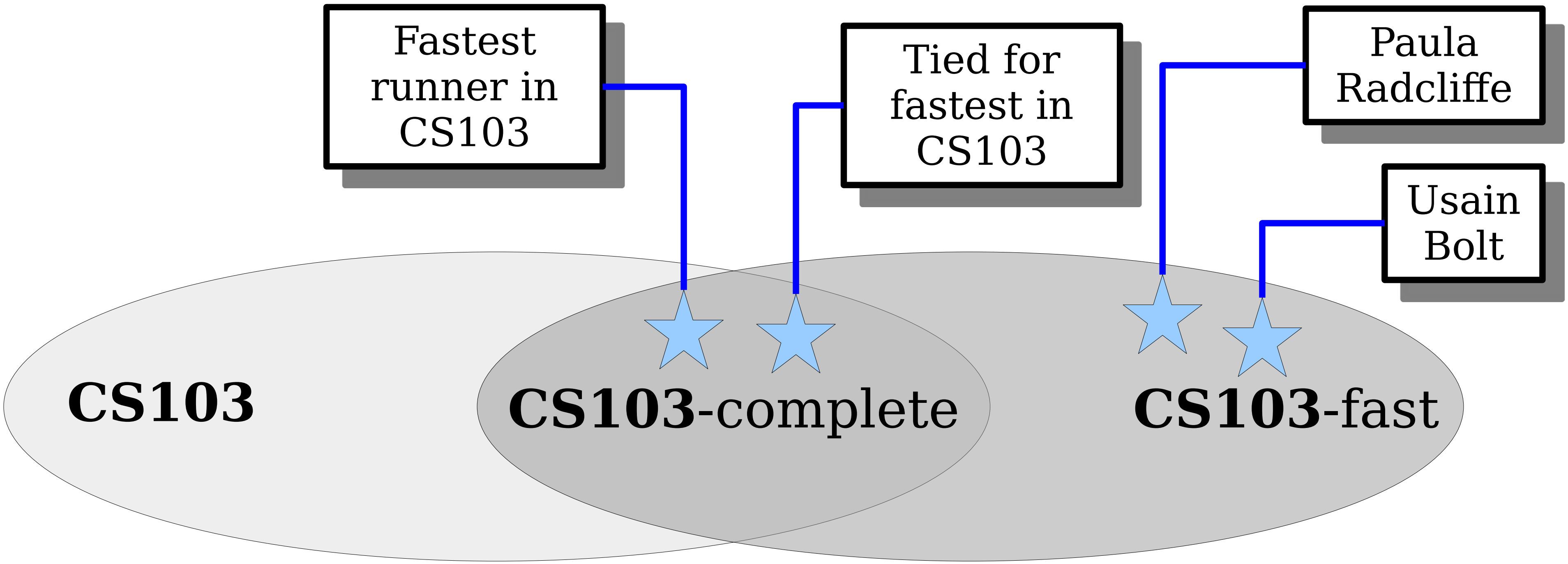
Most problems that we know are undecidable are either questions about what programs will and won't do (comes up in compilers, computer security, program verification, etc.), questions about mathematical proofs (automated theorem proving, etc.), or something in that space. There are problems we know of that are undecidable that don't look like these, but they're more the exception than the rule. It's pretty rare to bump into one of these problems outside of the contexts mentioned above.

Typically, the main concern you need to worry about is not undecidability but NP-hardness. What's NP-hardness? Well…

Back to CS103!

# **NP**-Hardness and **NP**-Completeness

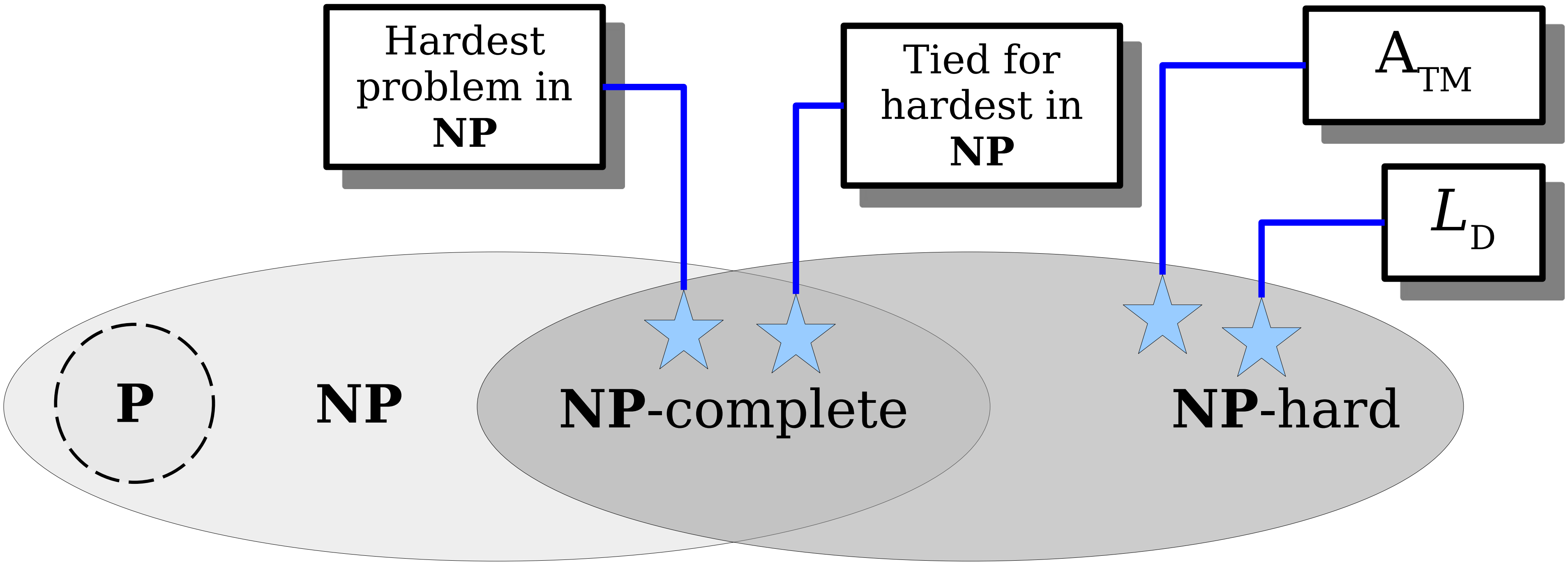
# ***An Analogy:*** Running Really Fast




For people $A$ and $B$, we say ***A* ≤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
$\forall A \in \mathbf{CS103}.\ A \leq_r P.$
***(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 $A \leq_p B$ 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 **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

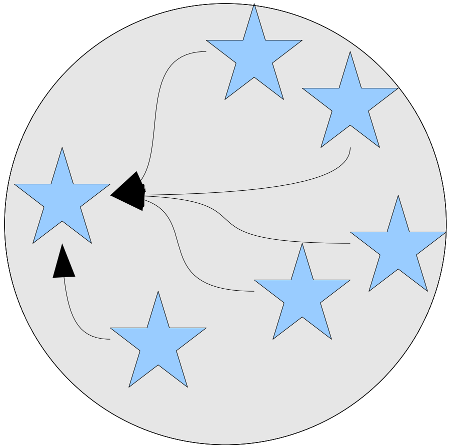
***Theorem:*** If *any* **NP**-complete language is in **P**, then **P = NP**.

***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** $\subseteq$ **P**, so **P** = **NP**. ■



P = NP

# The Tantalizing Truth

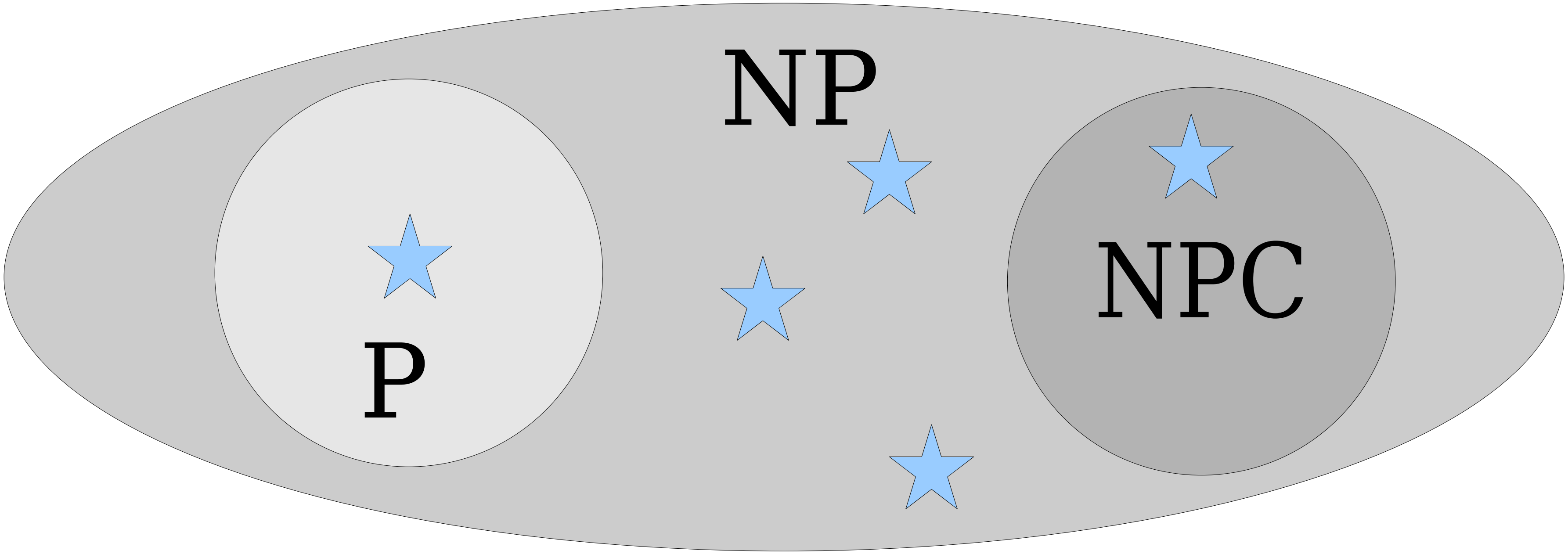
***Theorem:*** If *any* **NP**-complete language is not in **P**, then **P** ≠ **NP**.

***Intuition:*** This means the hardest problems in **NP** are so hard that they can't be solved in polynomial time. So the hardest problems in **NP** aren't in **P**, meaning **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?***

# Satisfiability

- A propositional logic formula φ is called ***satisfiable*** if there is some assignment to its variables that makes it evaluate to true.
  - $p \land q$ is satisfiable.
  - $p \land \neg p$ is unsatisfiable.
  - $p \to (q \land \neg q)$ is satisfiable.
- 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:

***SAT*** **= { ⟨φ⟩ | φ is a satisfiable PL formula }**

***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: have the certificate be a 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 is 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!

# 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.

# 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)*

***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!***