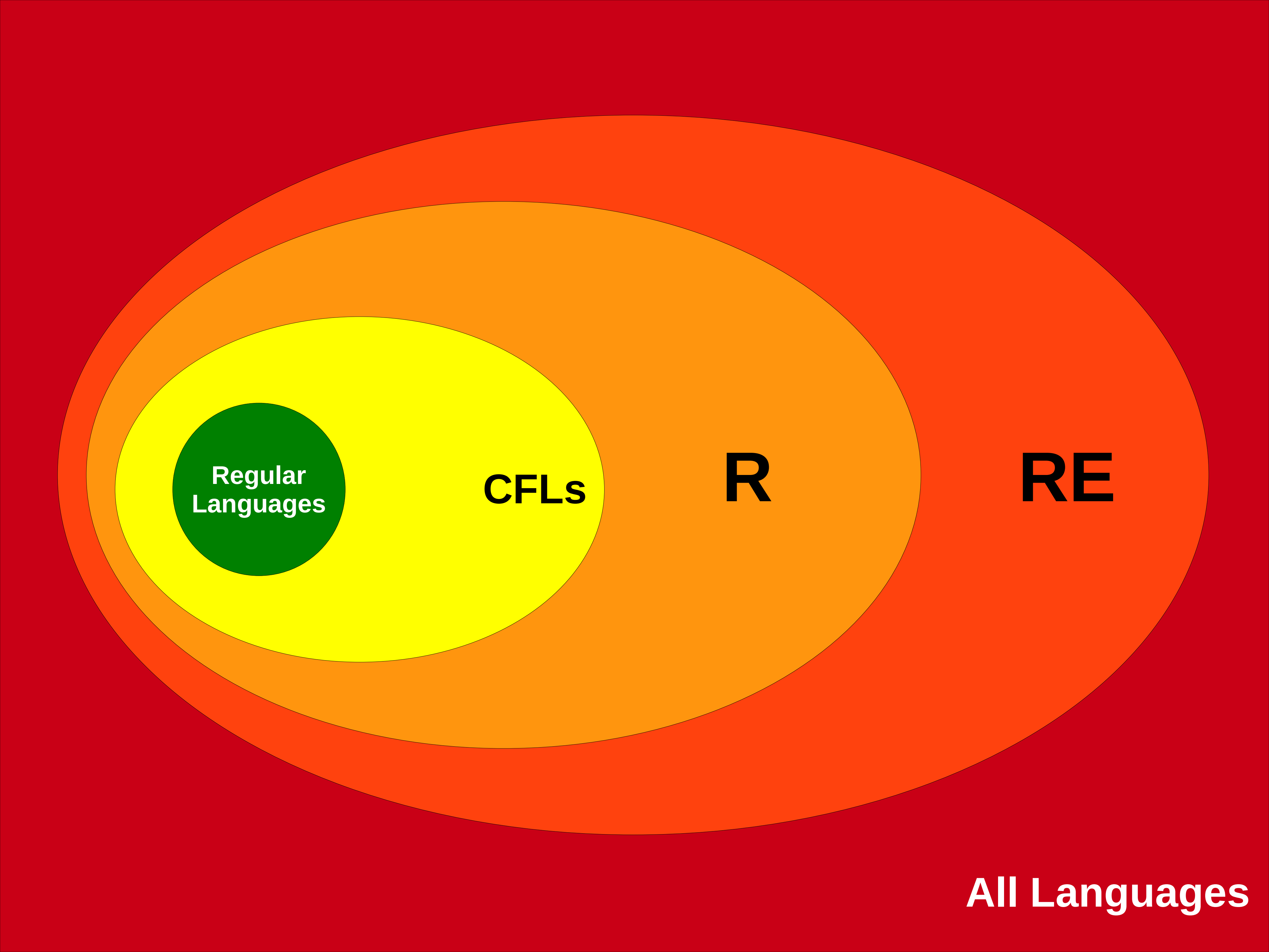
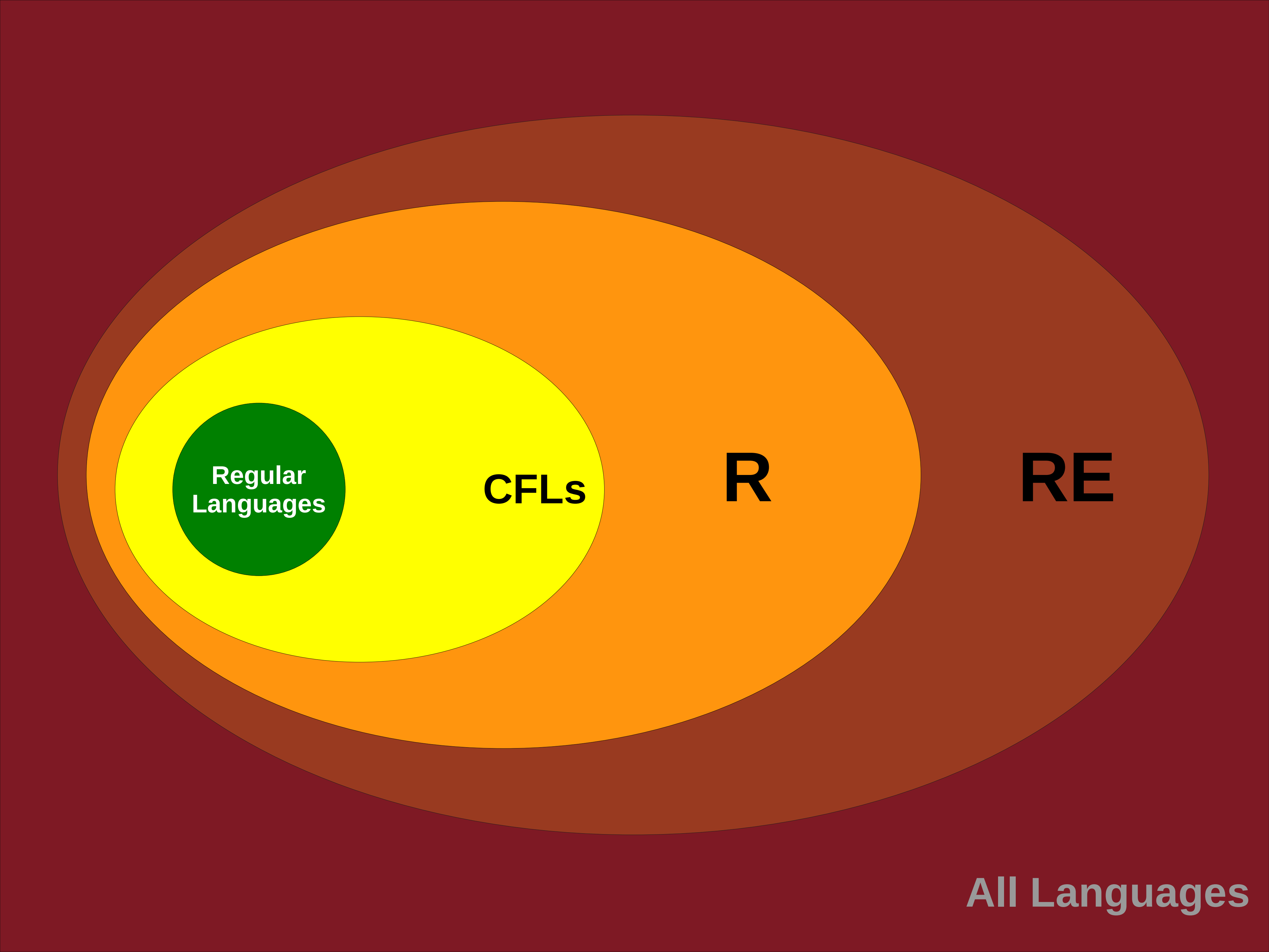
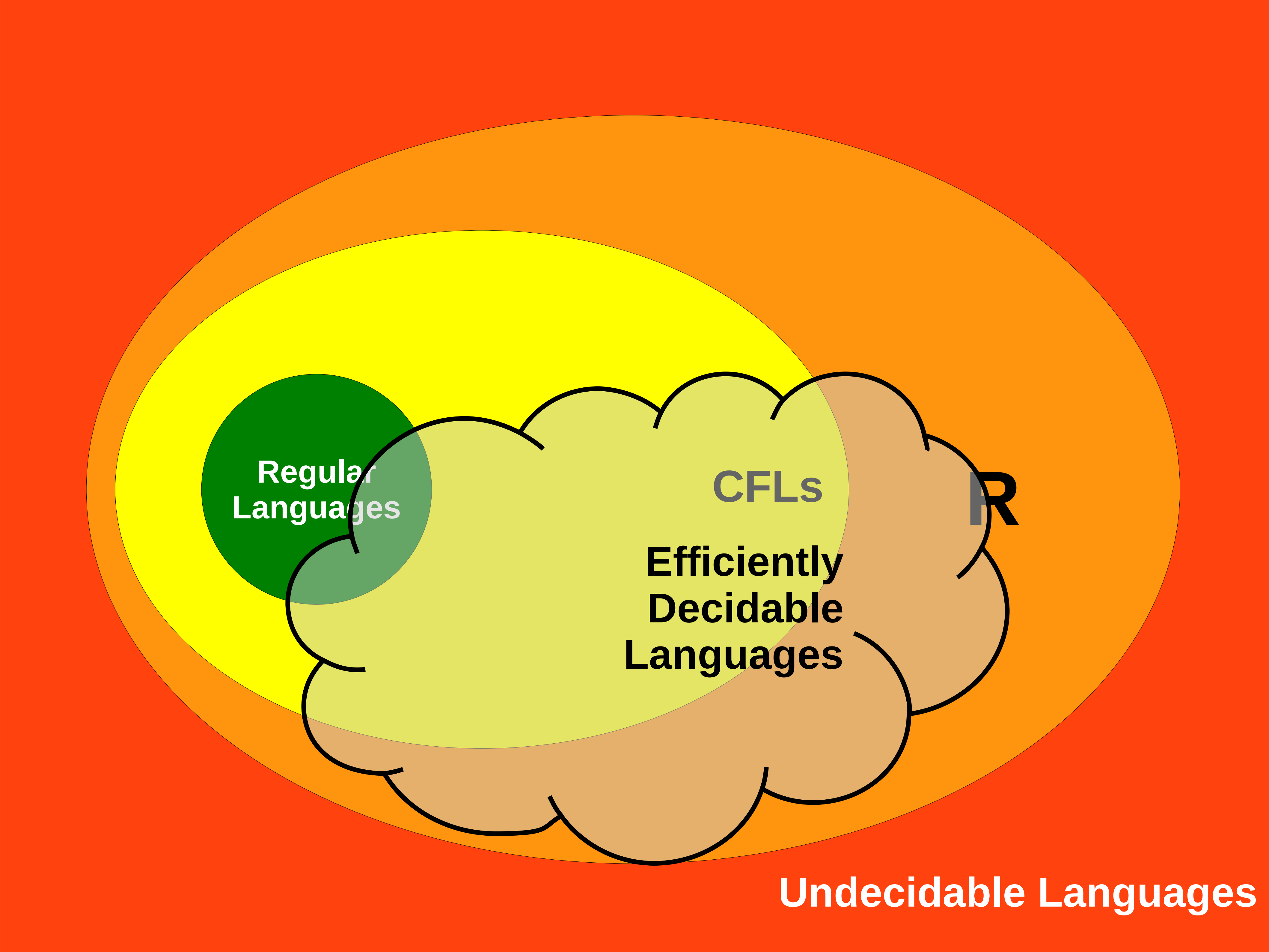
CS103
Winter 2025

Supplemental:

Complexity Theory Recap


Regular Languages
CFLs
R
RE
All Languages


Regular
Languages
CFLs
R
RE
All Languages


Regular Languages
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

A Sample Problem

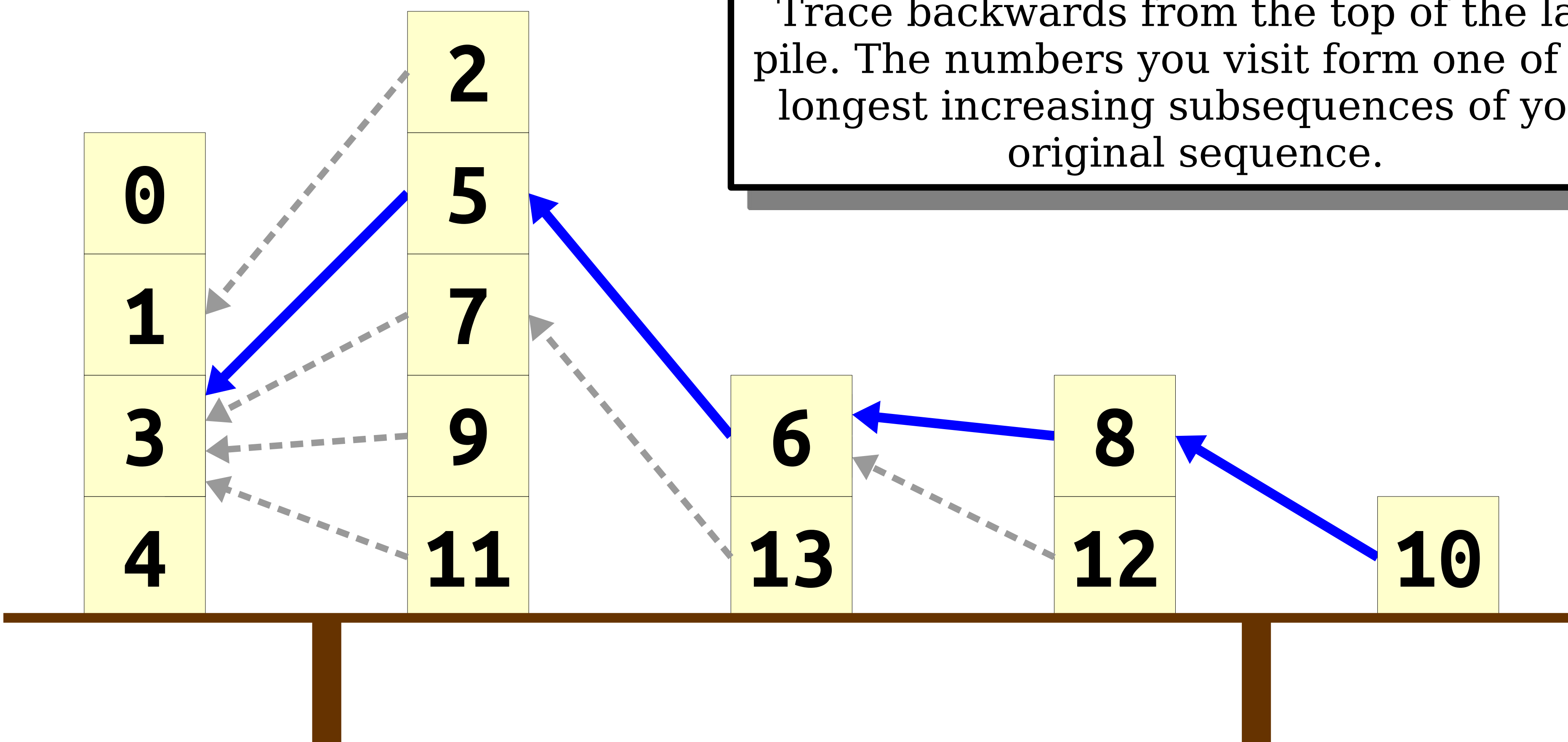
4	3	11	9	7	13	5	6	1	12	2	8	0	10

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

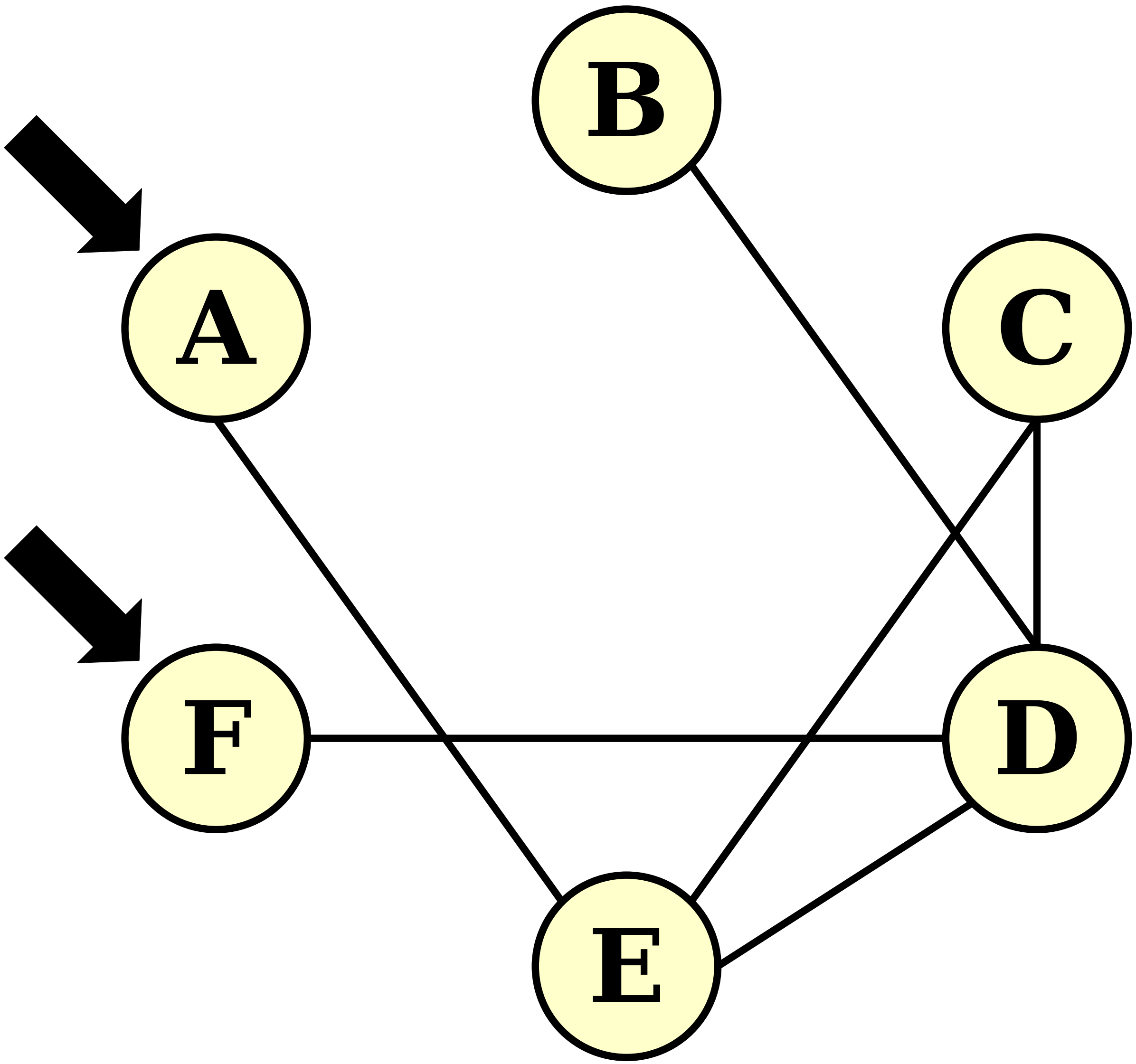
Patience Sorting

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

Trace backwards from the top of the last pile. The numbers you visit form one of the longest increasing subsequences of your original sequence.

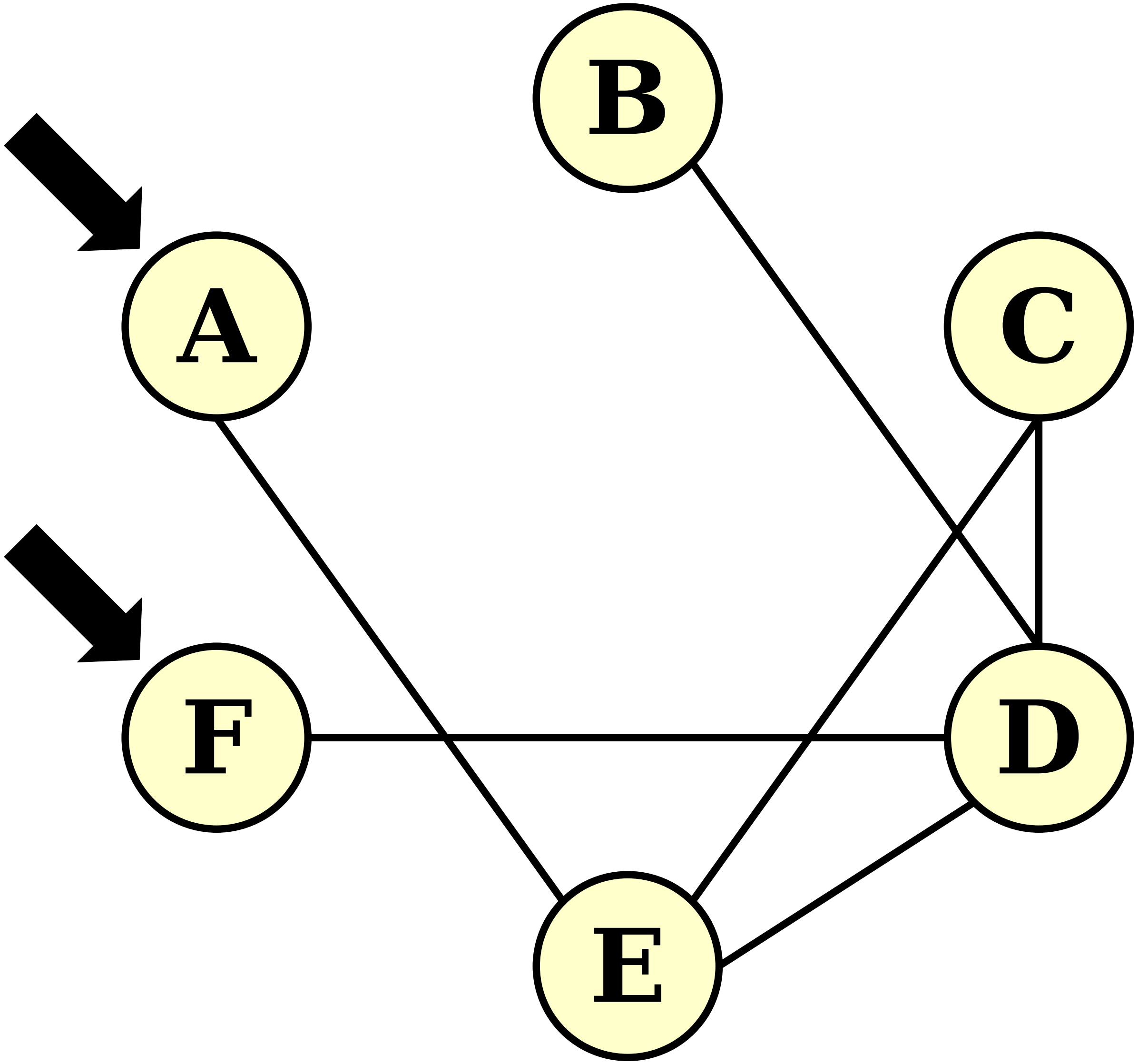


Another Problem



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

Another Problem



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

Idea: Use BFS!

For Comparison

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

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.

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**

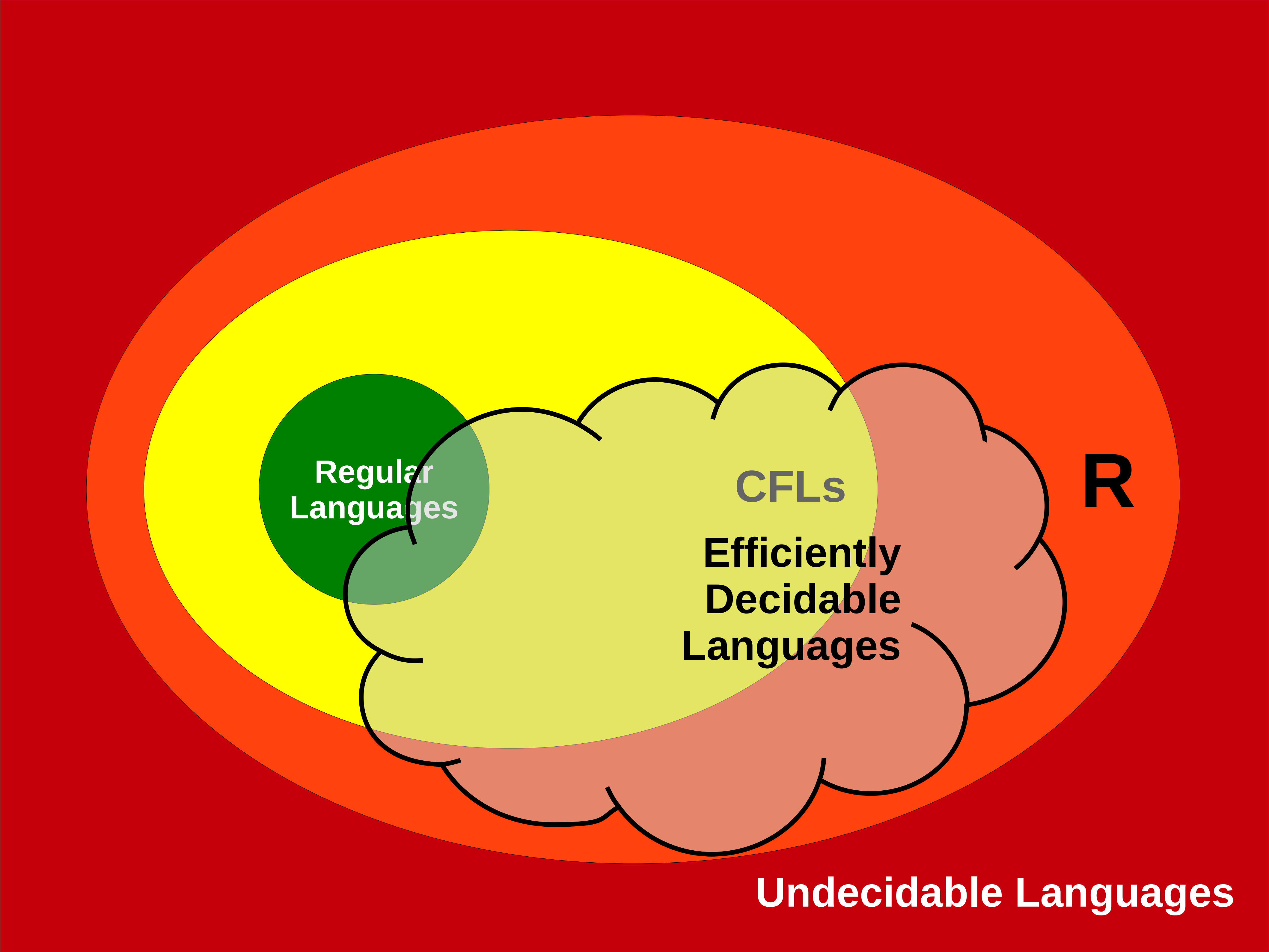
- The ***complexity class*** **P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

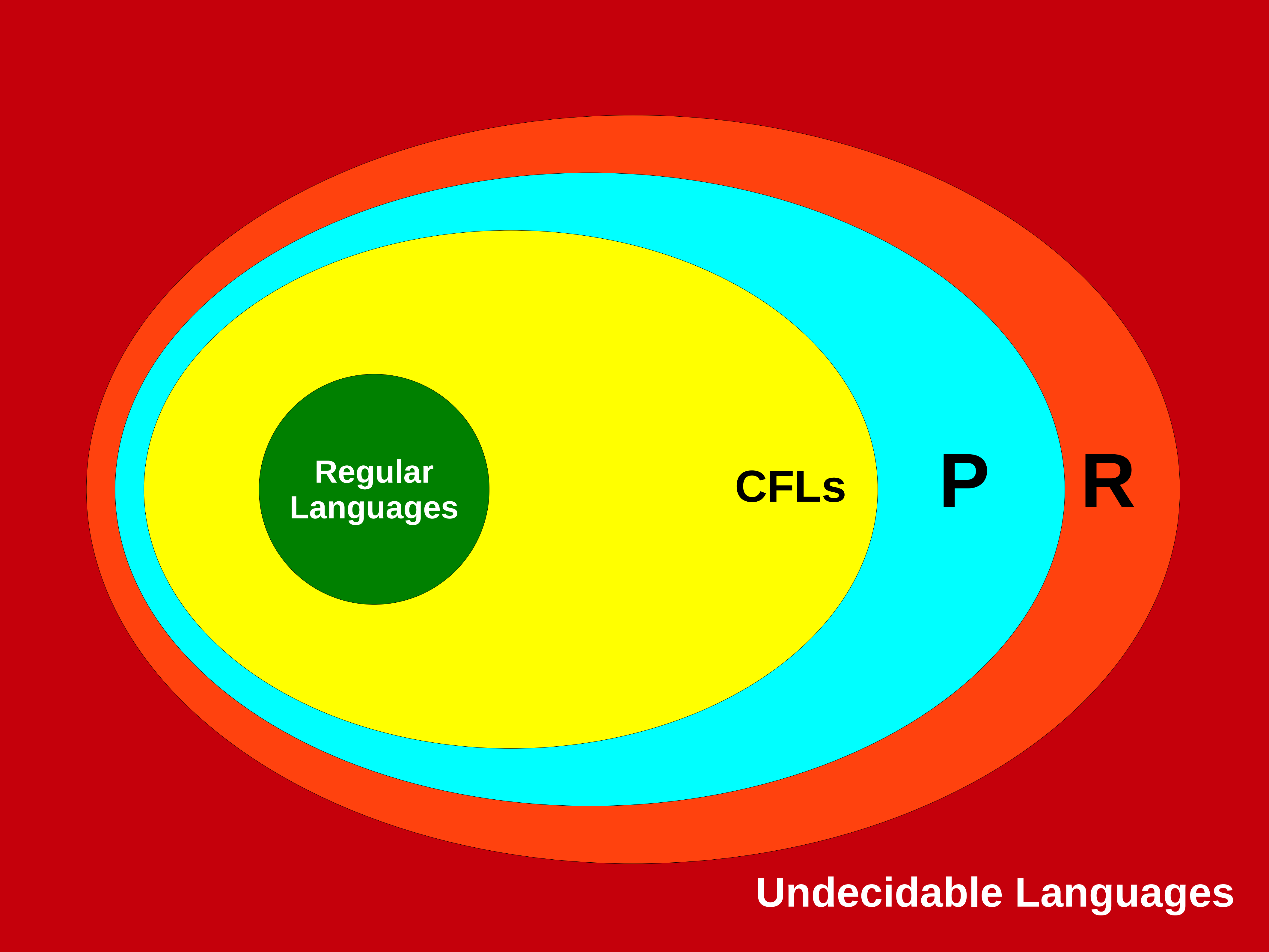
  P = { *L* | There is a polynomial-time decider for *L* }

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

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

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

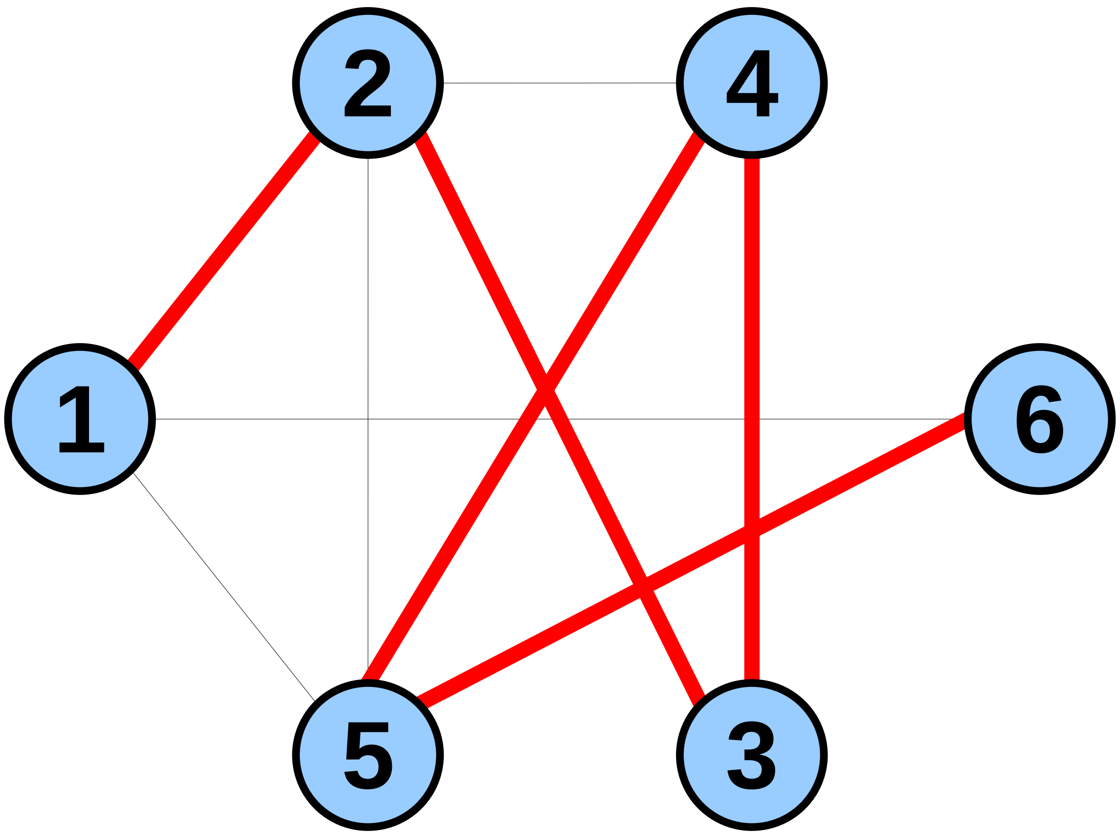
Does this Sudoku problem have a solution?

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 a path that goes through
every node exactly once?

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:

NP = { L | 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** $\subsetneq$ **R**.
  - ***Proof idea:*** If $L \in$ **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.)

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

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

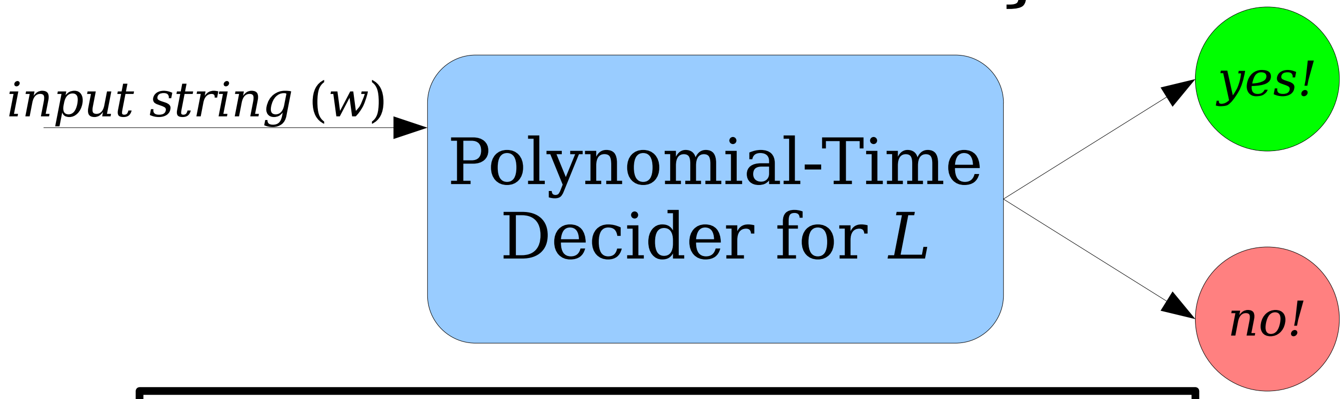
$$\mathbf{R} = \{ L \mid \text{there is a } \sout{\text{polynomial-time}} \text{ decider for } L \}$$

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

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

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

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

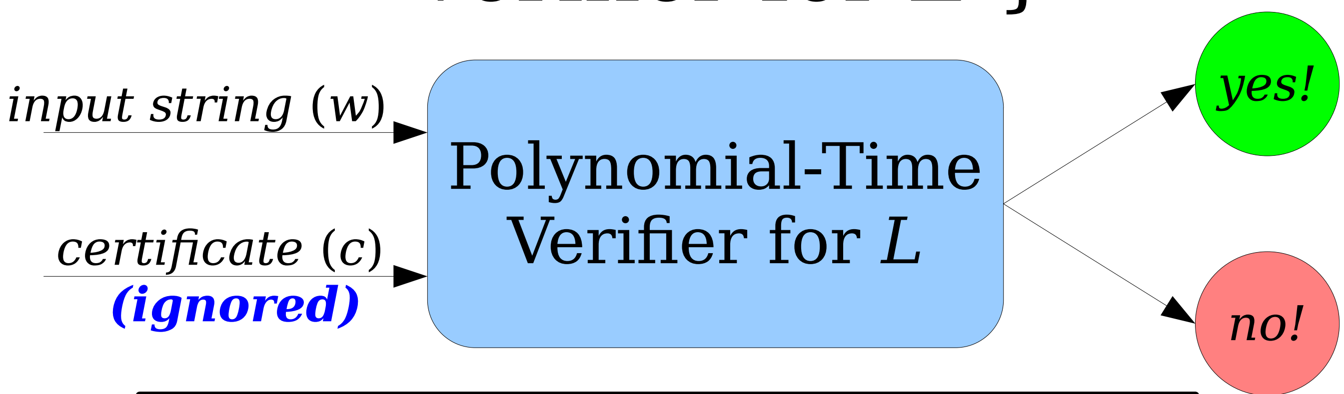



```
bool solveProblemL(string w) {

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

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

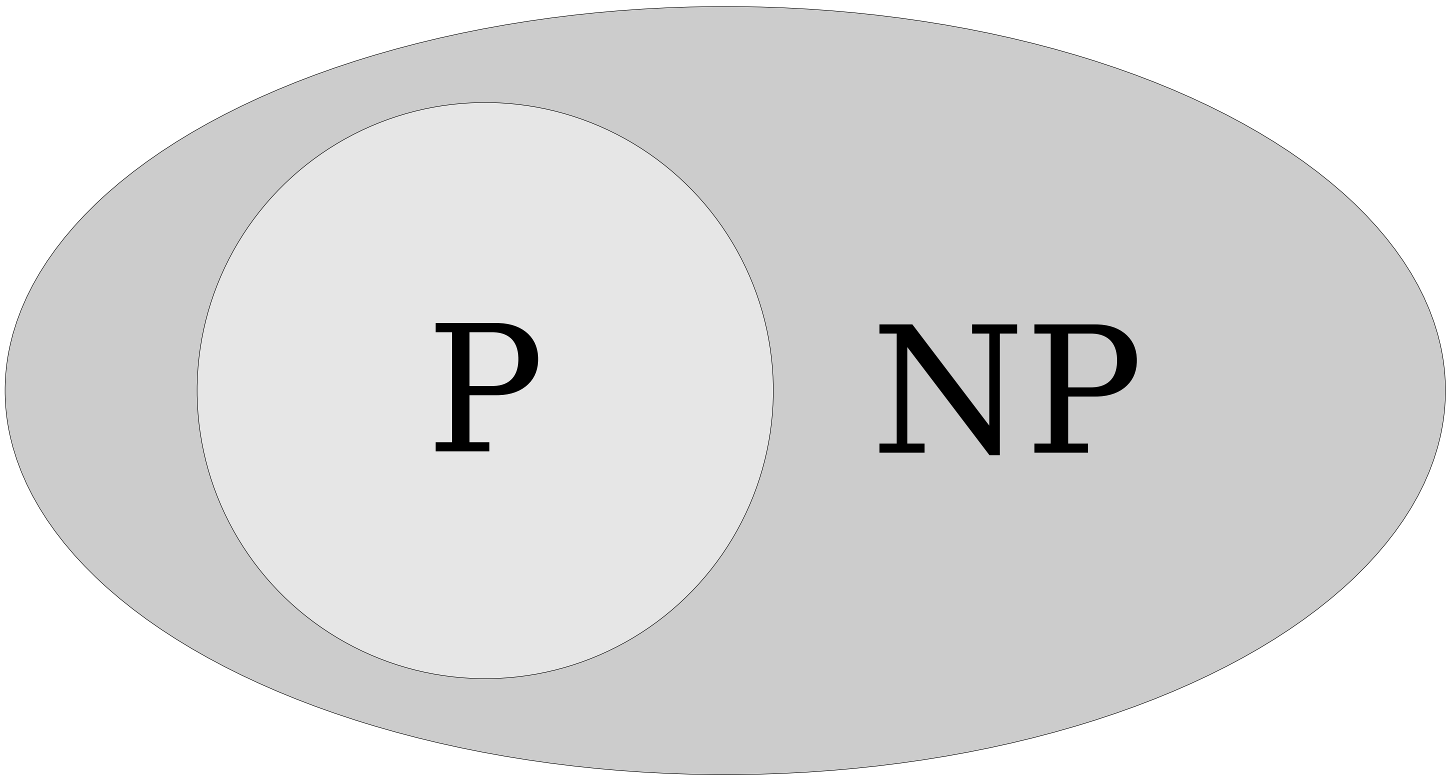
NP = { L | There is a polynomial-time verifier for L }



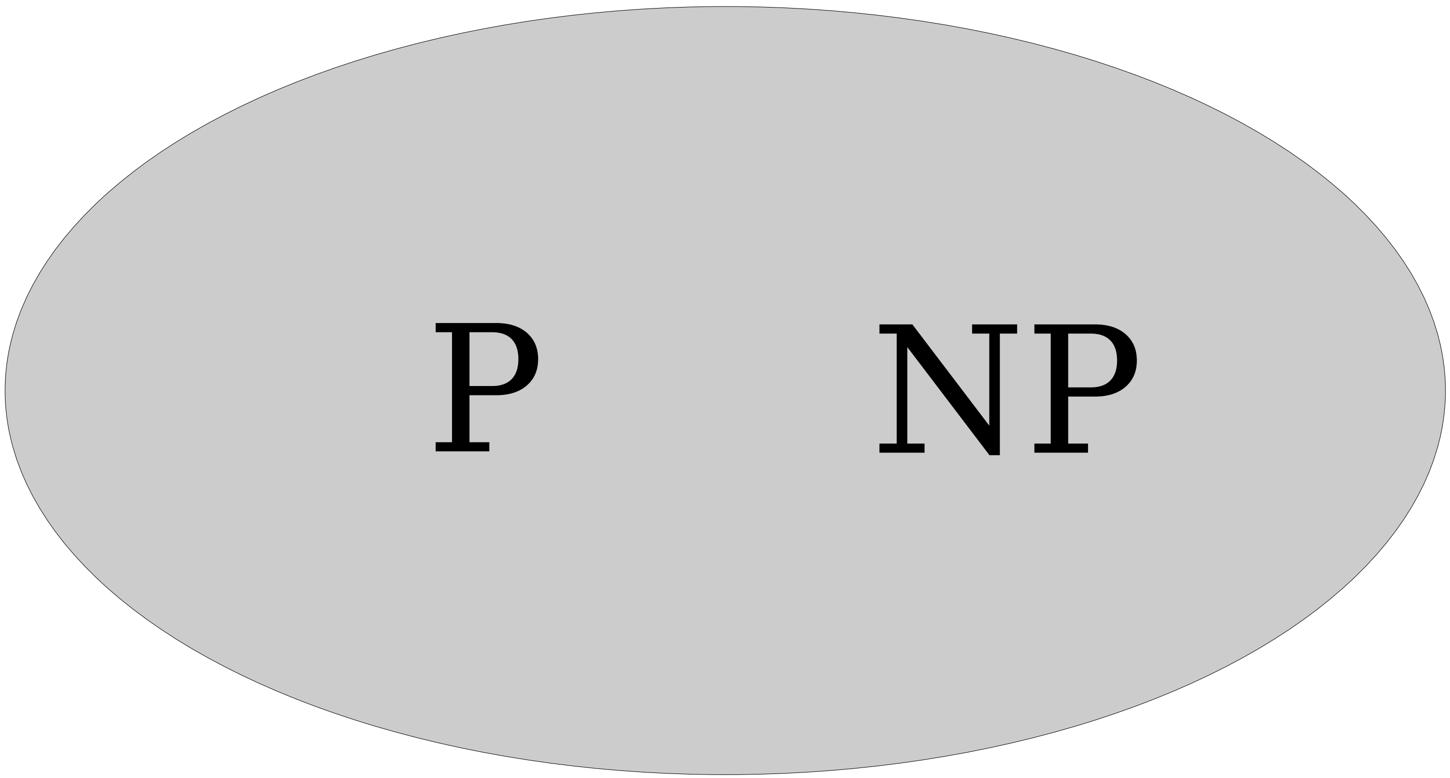

```
bool solveProblemL(string w, string c) {
    /* don't even look at c */
    do some work;
    return the answer;
}
```

$$\mathbf{P} \subseteq \mathbf{NP}$$

Which Picture is Correct?



Which Picture is Correct?

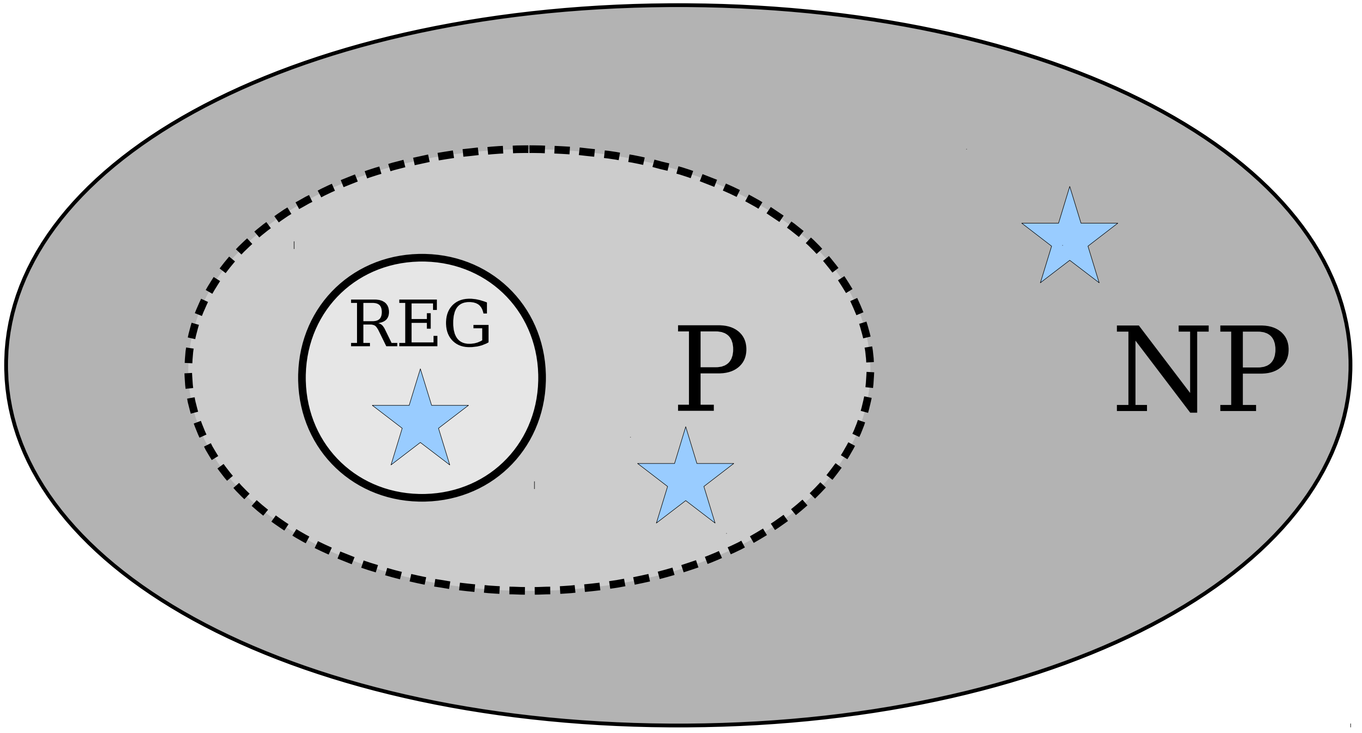


A Problem

- 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!

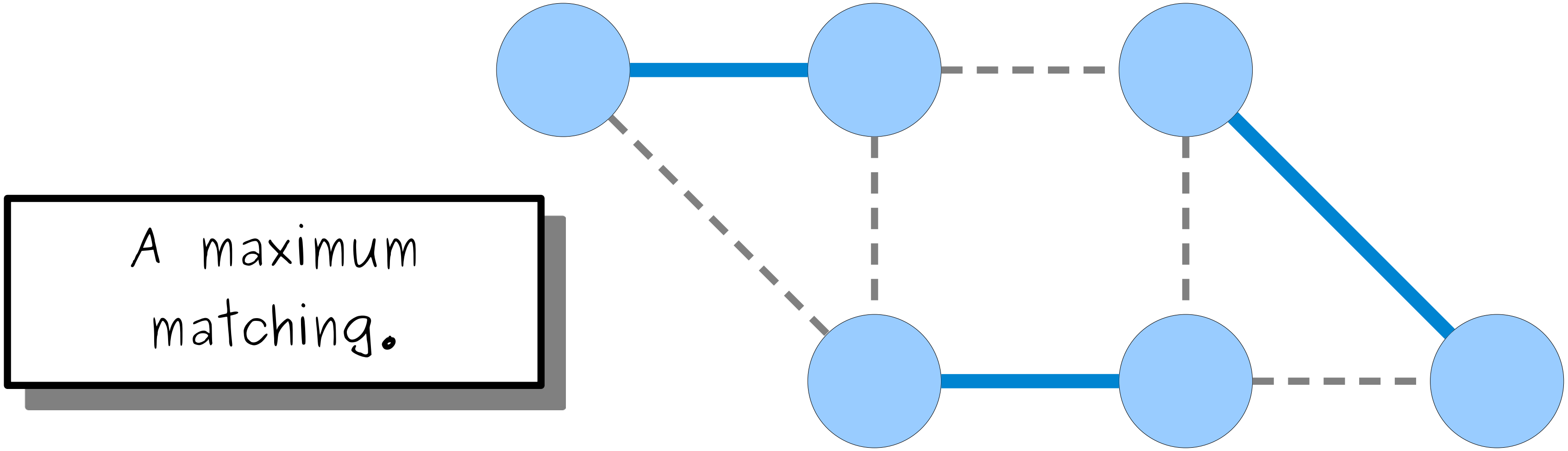


Problems in **NP** vary widely in their difficulty, even if **P** = **NP**.

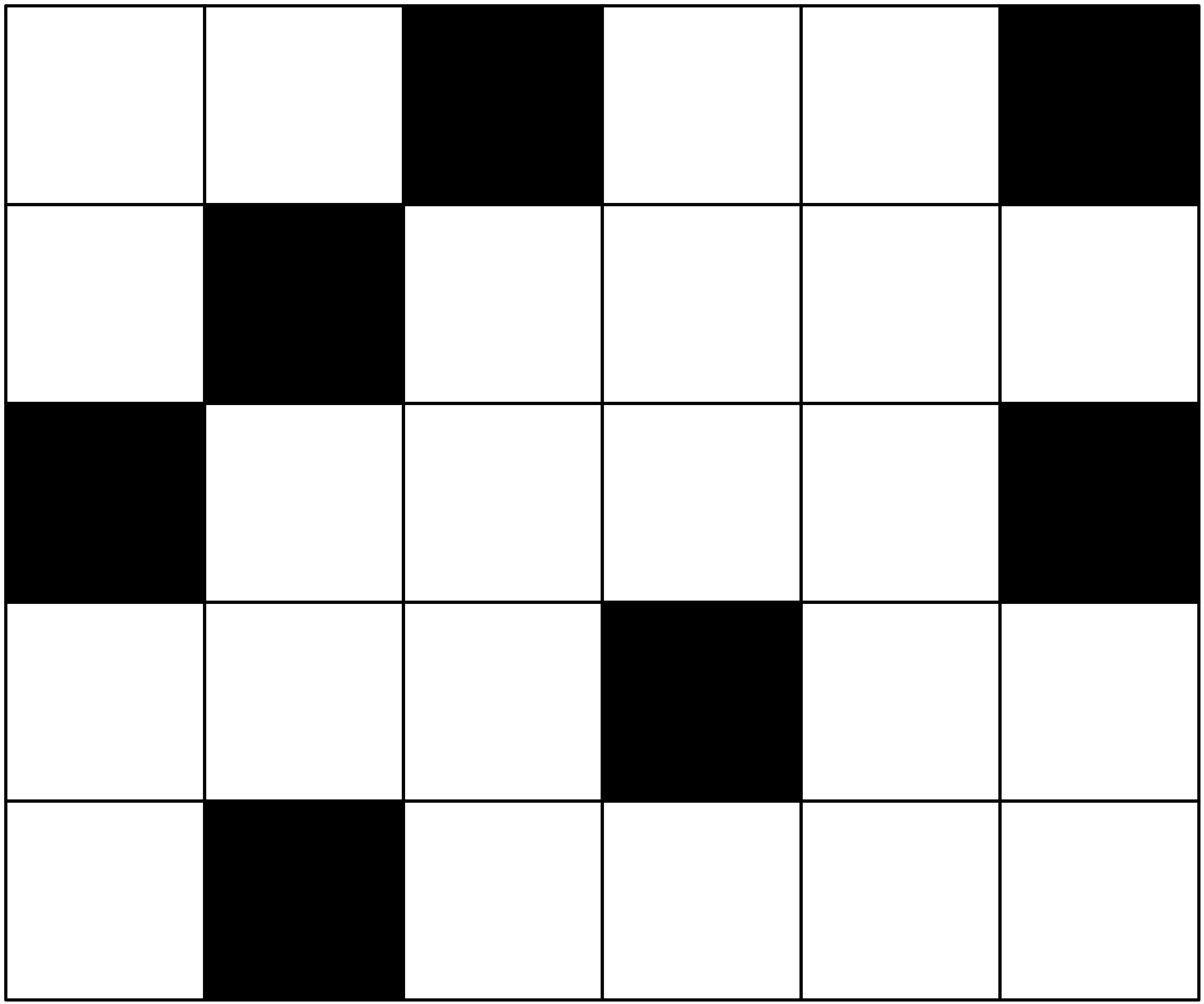
How can we rank the relative difficulties of problems?

Maximum Matching

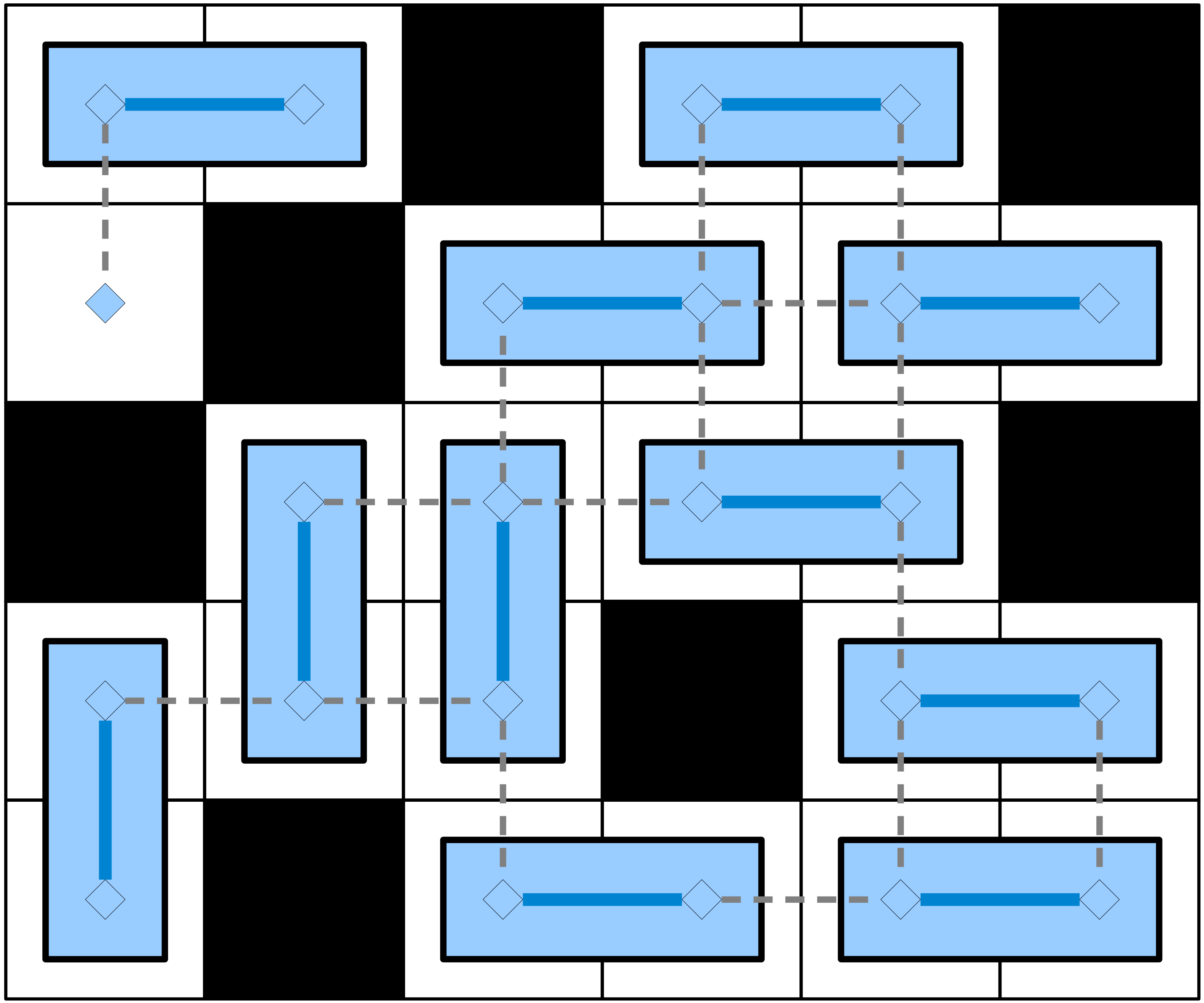
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



Solving Domino Tiling



Solving Domino Tiling



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

$$\mathit{DominoTiling} \leq_p \mathit{MaximumMatching}$$

- We say that ***Domino Tiling is polynomial-time reducible to Maximum Matching***
- Maximum Matching is at least as hard as Domino Tiling.

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

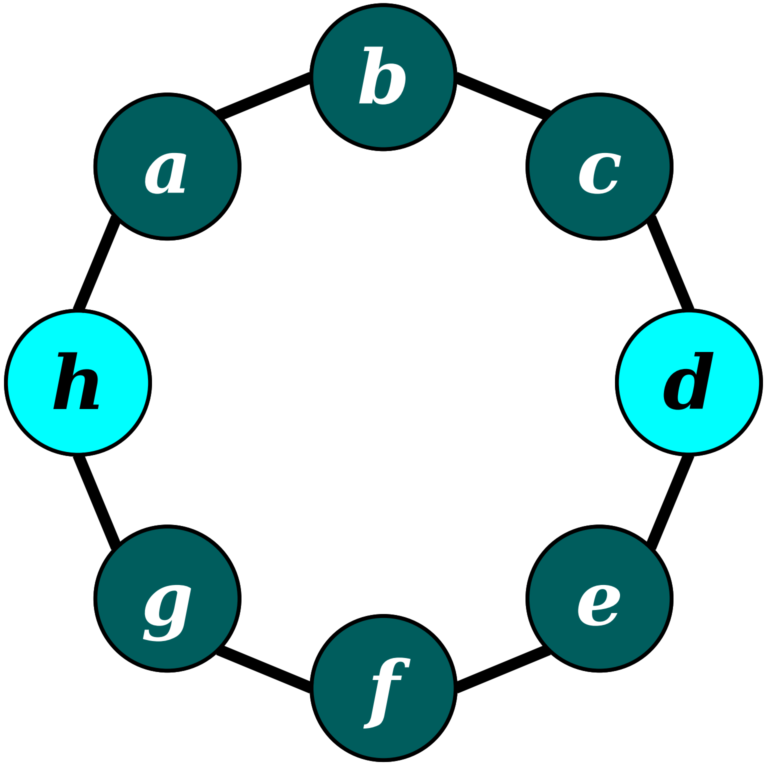
- The ***boolean satisfiability problem*** (***SAT***) is the following:

Given a propositional logic formula φ, is φ satisfiable?

- Formally:

SAT **= { ⟨φ⟩ | φ 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?



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

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

Observation 2: Button press order doesn't matter.

Observation 3: Our propositional formula will have one variable per button, indicating whether we press it.

Observation 4: A light that is initially off stays off when an even number of adjacent lights are pressed.

Observation 5: A light that is initially on ends off when an odd number of adjacent lights are pressed.

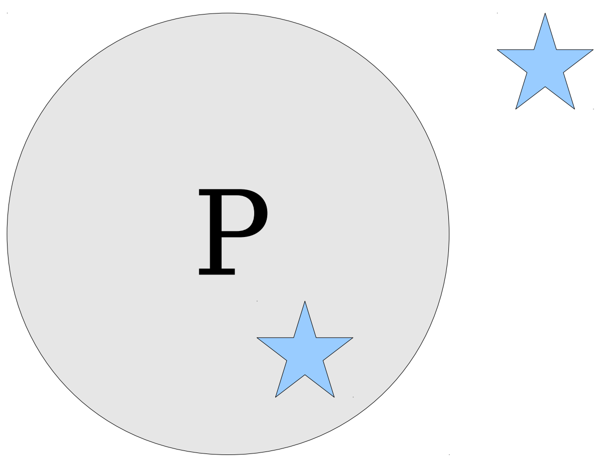
```
bool canTurnLightsOff(LightRing r) {
    return isSatisfiable(ringToFormula(r));
}
```

$$LightsOut \leq_p SAT$$

- We say that ***Lights Out is polynomial-time reducible to SAT***
- SAT is at least as hard as Lights Out.

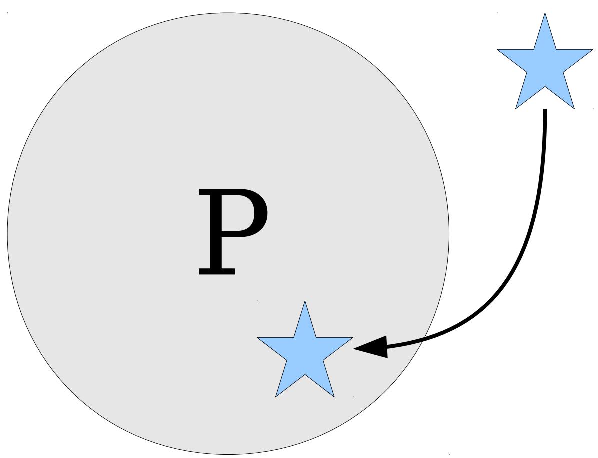
Polynomial-Time Reductions

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



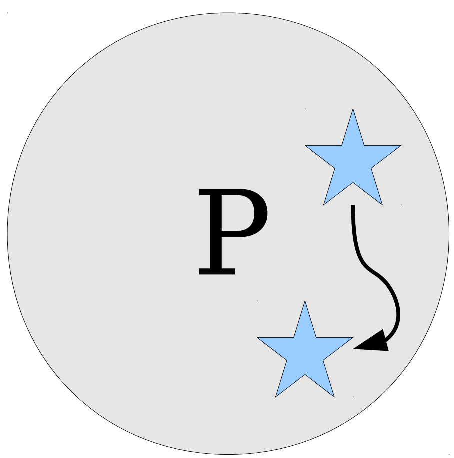
Polynomial-Time Reductions

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



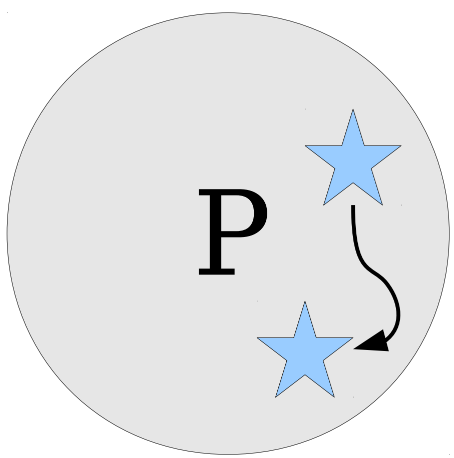
Polynomial-Time Reductions

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



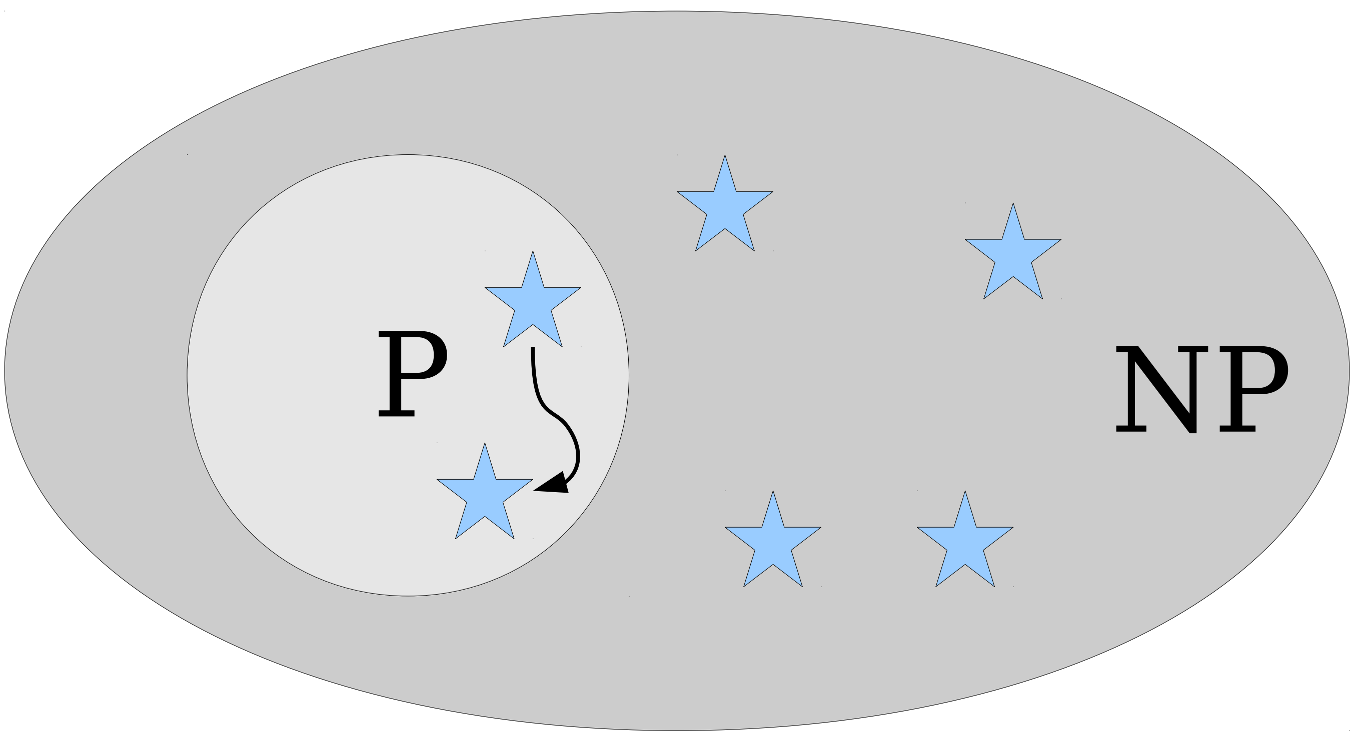
Polynomial-Time Reductions

- 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 $A \in \mathbf{NP}$.



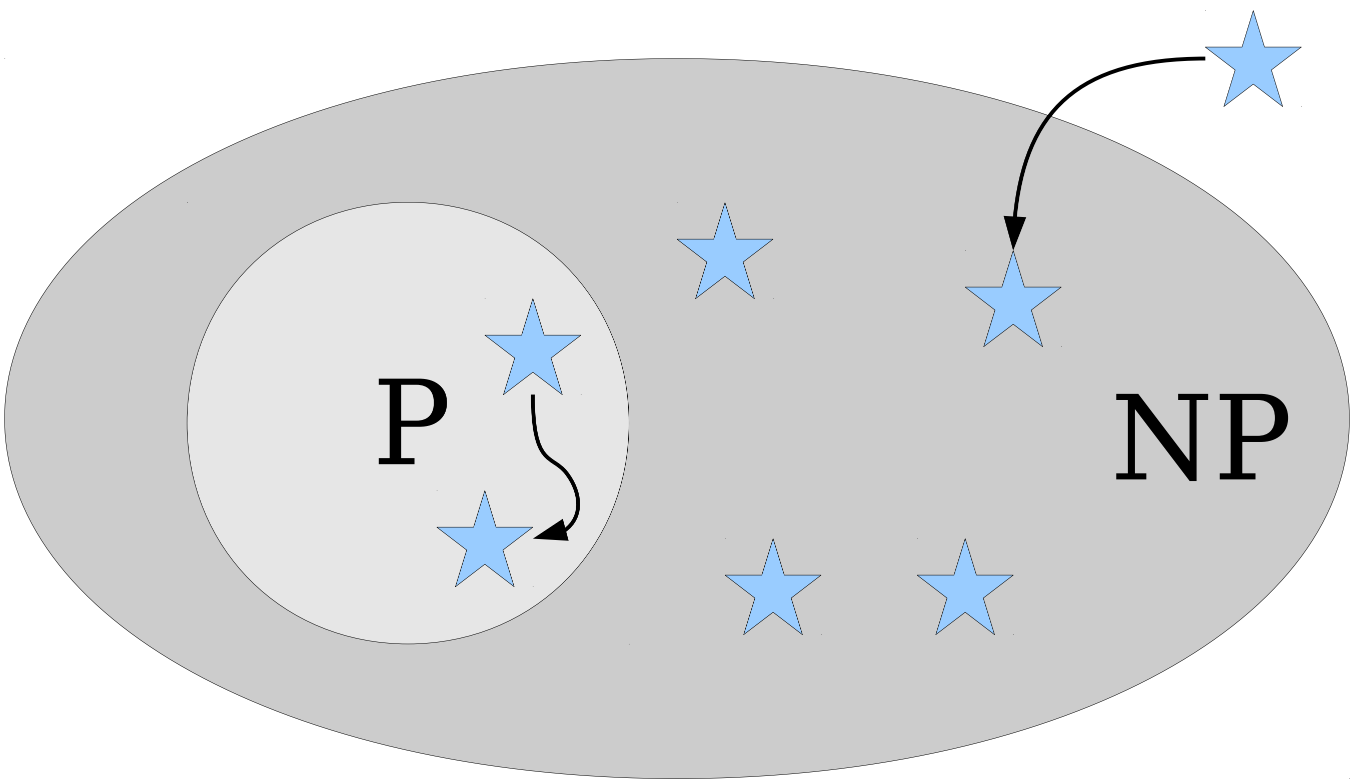
Polynomial-Time Reductions

- 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 $A \in \mathbf{NP}$.



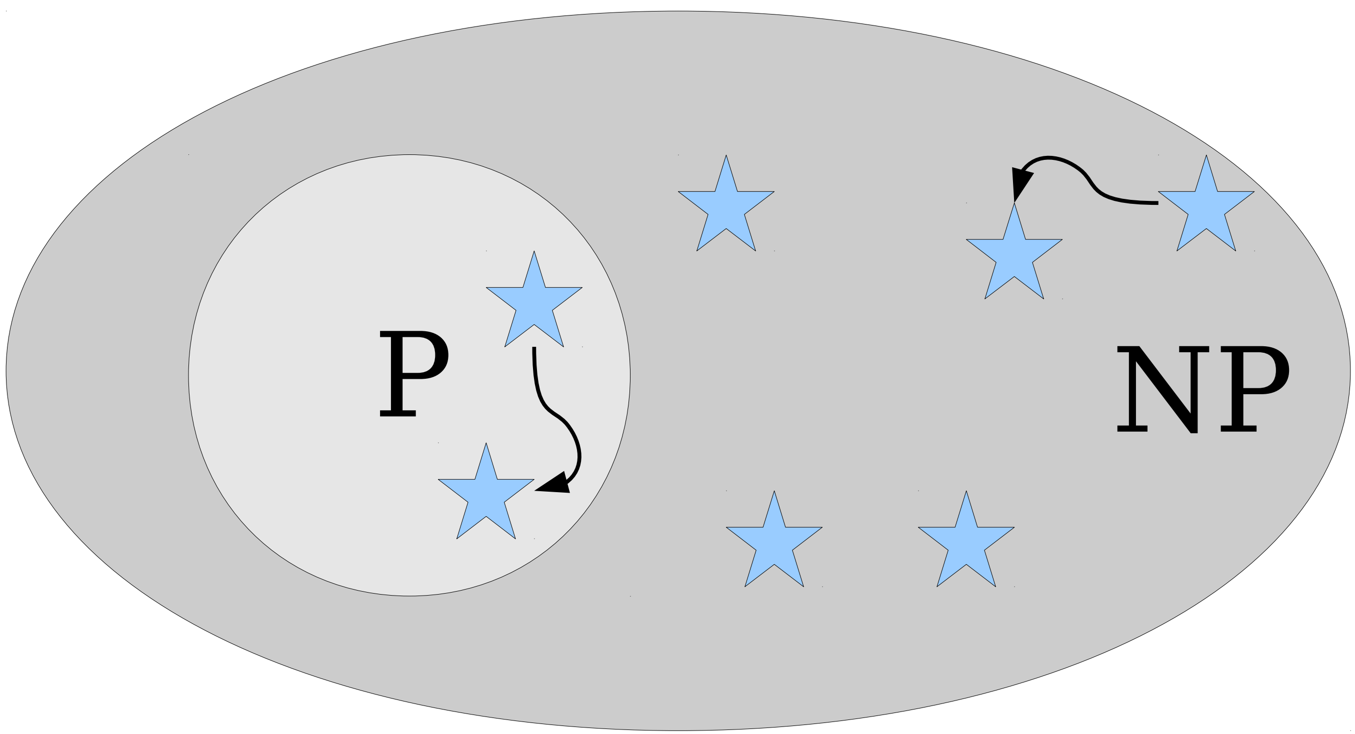
Polynomial-Time Reductions

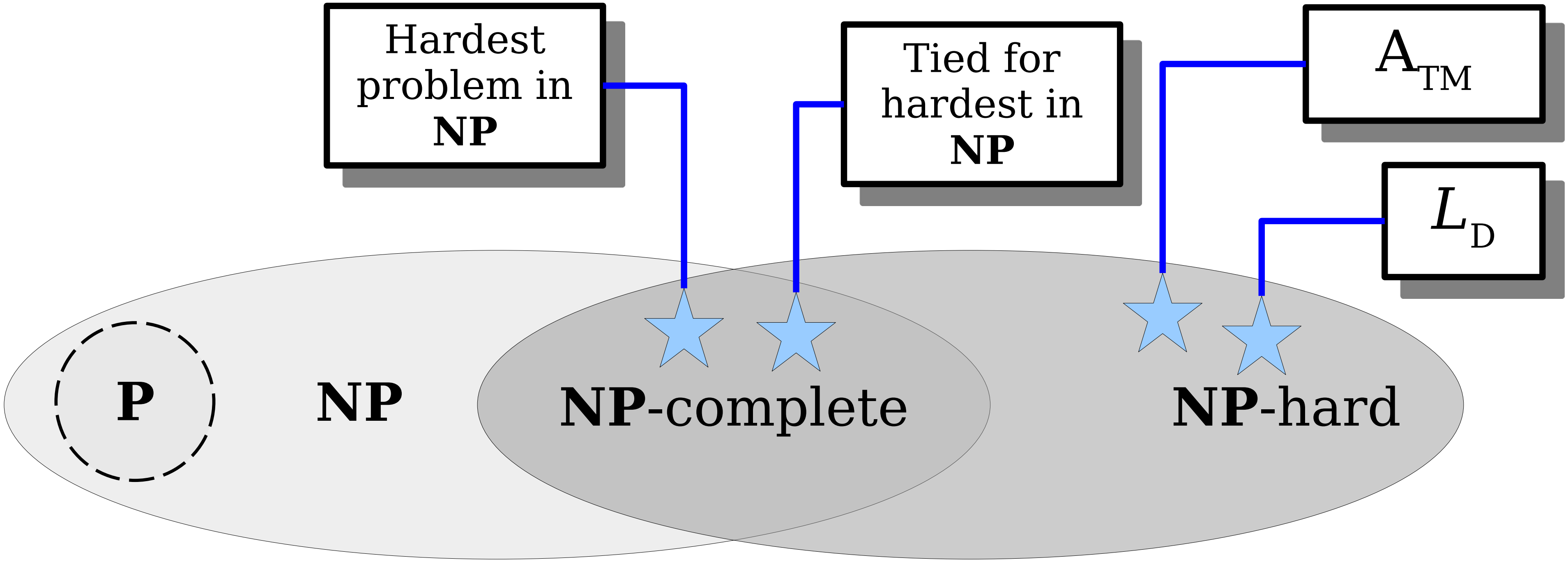
- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



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 $A \in \mathbf{NP}$.





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

$L \in$ **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.

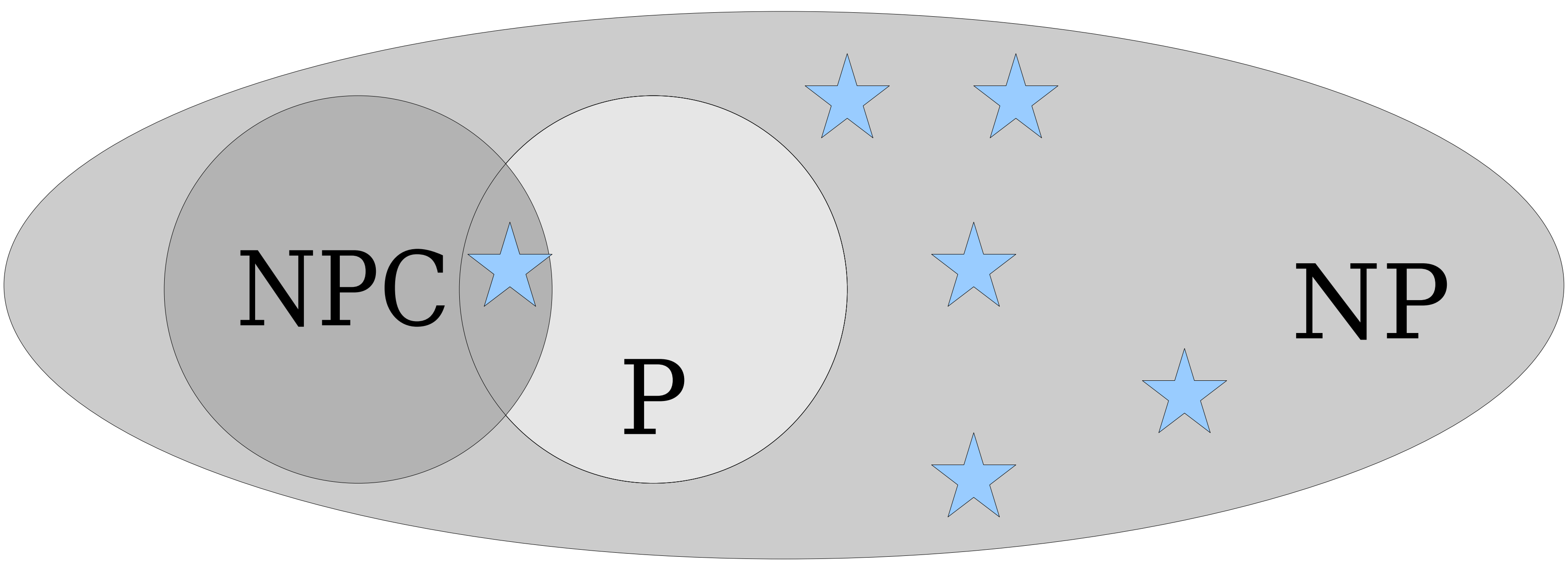
The Tantalizing Truth

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.

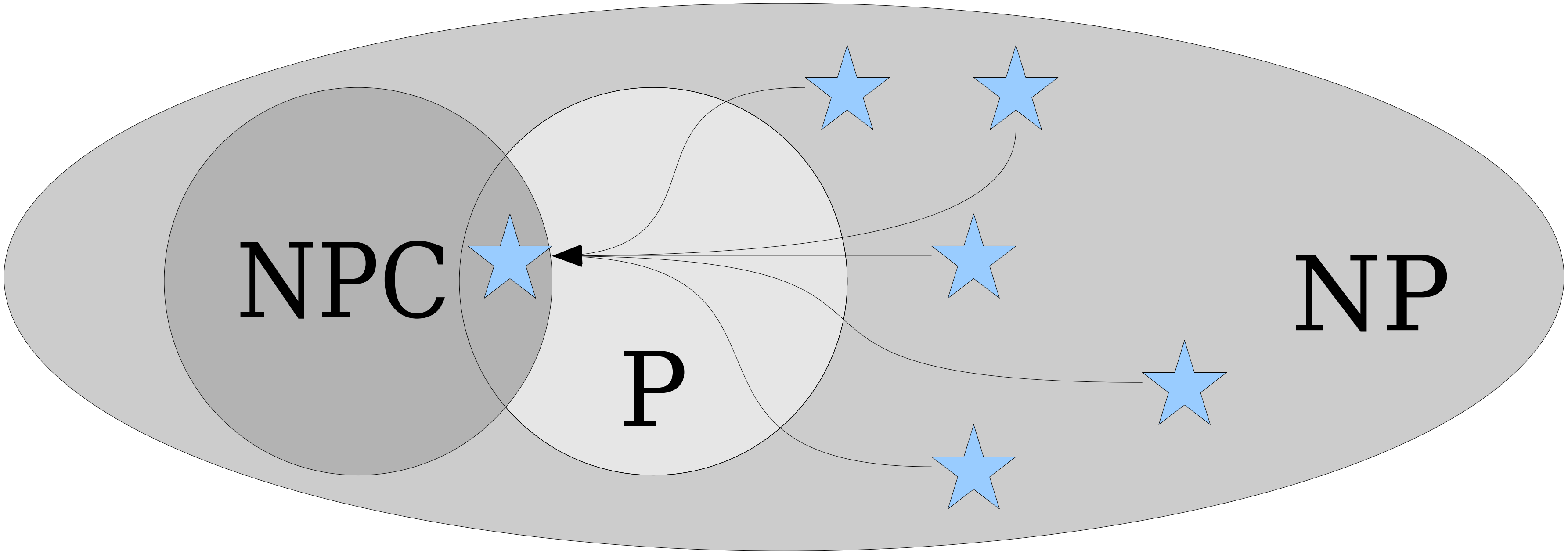
The Tantalizing Truth

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



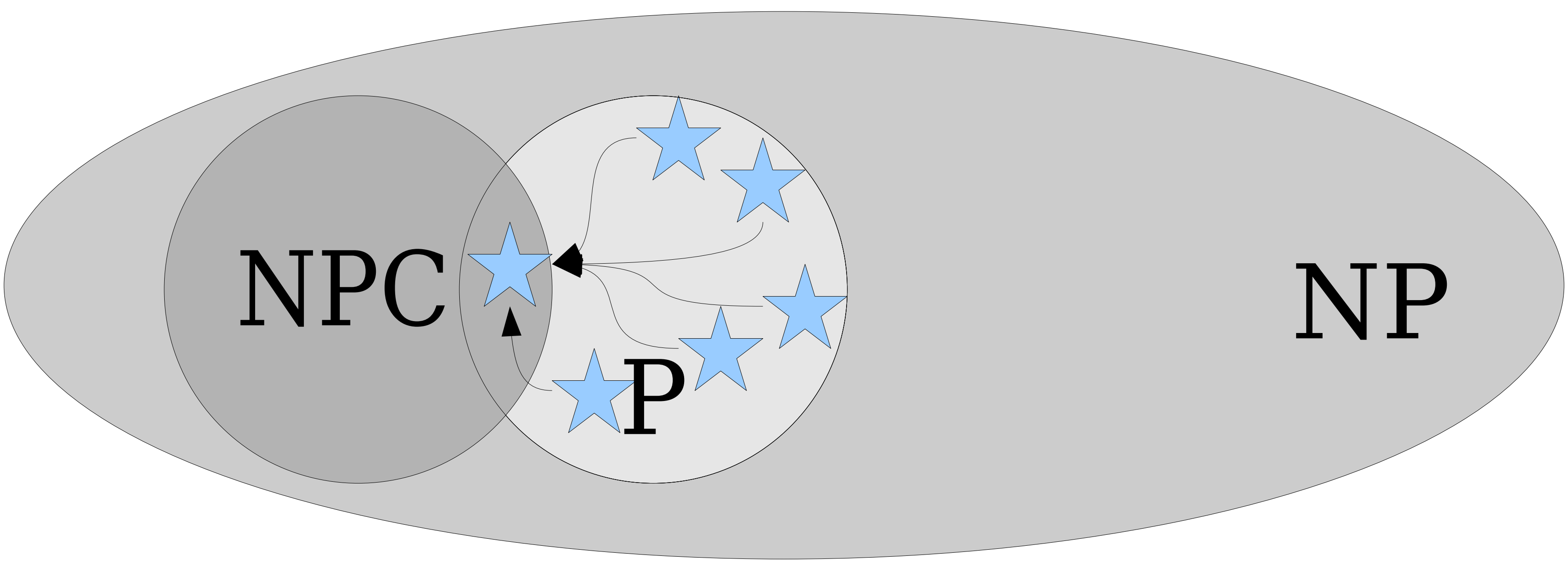
The Tantalizing Truth

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



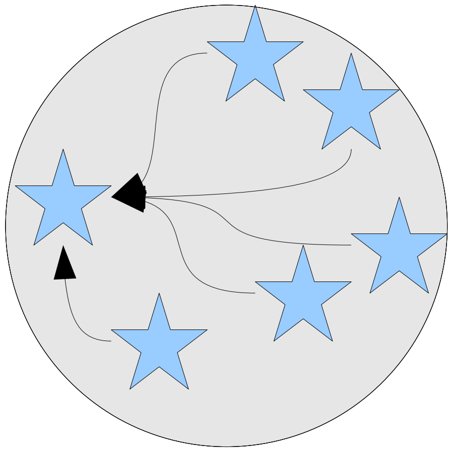
The Tantalizing Truth

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



The Tantalizing Truth

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

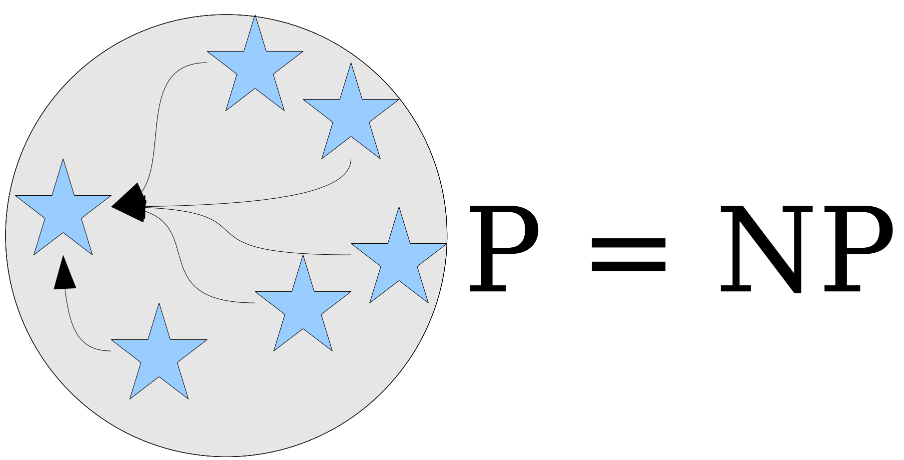


P = NP

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**. ■



The Tantalizing Truth

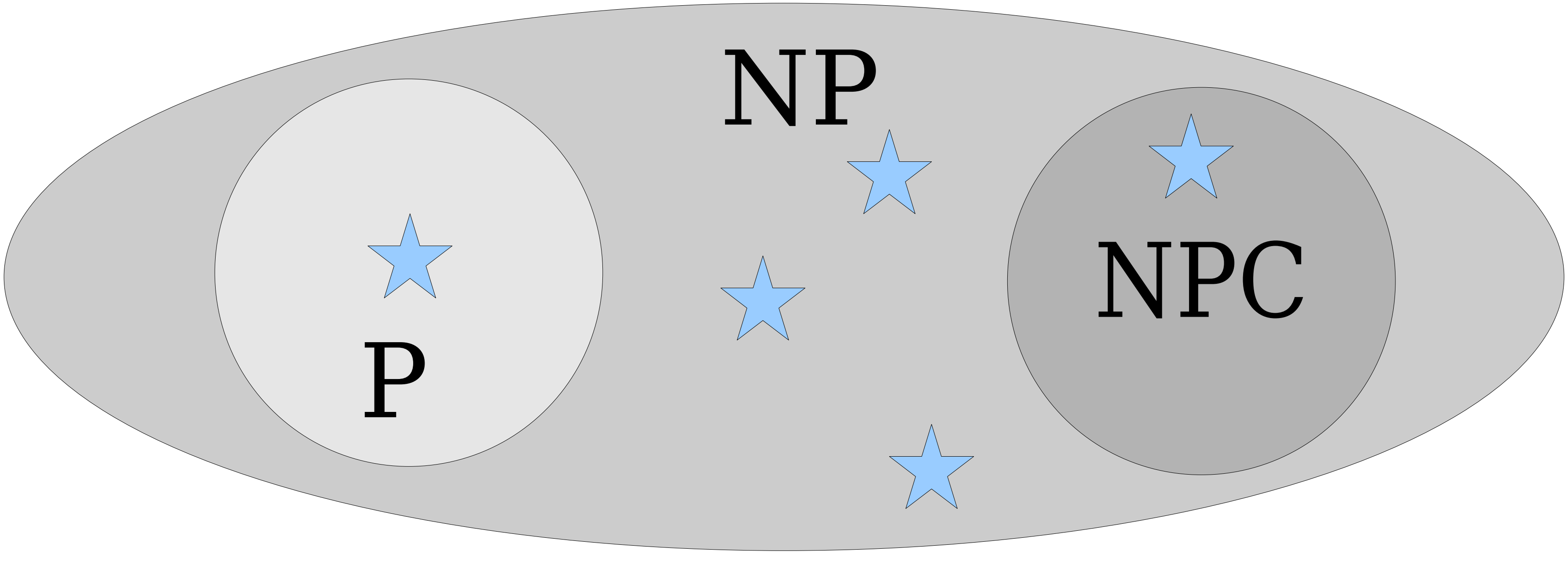
Theorem: If *any* **NP**-complete language is not in **P**, then **P** ≠ **NP**.

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?

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 $\varphi(w)$ that says "there is a certificate c where V accepts $\langle w, c \rangle$." This formula is satisfiable if and only if $w \in L$, so deciding whether the formula is satisfiable decides whether w is in L. ■

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 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.

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.