

# Complexity Theory

Up to this point:

*“Can computers solve this problem?”*  
**(Computability Theory)**

Up to this point:

*“Can computers solve this problem?”*  
**(Computability Theory)**



Up to this point:

“*Can* computers solve this problem?”  
(**Computability Theory**)

Starting today:

“Ok, even if we *can*, we need to consider whether the time/resources required actually make practical/feasible sense.”  
(**Complexity Theory**)

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64			
7	128			
8	256			
9	512			
10	1,024			
30	2,070,000,000			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64			
7	128			
8	256			
9	512			
10	1,024			
30	2,070,000,000			

2.4s

Easy!

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128			
8	256			
9	512			
10	1,024			
30	2,070,000,000			

**194 YEARS**

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256			<b>3.59E+21 YEARS</b>
9	512			
10	1,024			
30	2,070,000,000			



$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512			
10	1,024			
30	2,070,000,000			

For comparison: there are about  $10E+80$  atoms in the universe. No big deal.

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024			<b>1.42E+137 YEARS</b>
30	2,070,000,000			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
30	2,070,000,000	64,062,560,941 (35s)	4,284,900,000,000,000,000 (75 years)	LOL

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
30	2,070,000,000	64,062,560,941 (35s)	4,284,900,000,000,000,000 (75 years)	$1.06 \times 10^{623,132,091}$

$1.86 \times 10^{623,132,074}$  years

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
30	2,070,000,000	64,062,560,941 (35s)	4,284,900,000,000,000,000 (75 years)	$1.06 \times 10^{623,132,091}$

$2^n$  is clearly infeasible, but look at  $\log_2 n$   
—only a tiny fraction of a second!

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
30	2,070,000,000	64,062,560,941 (35s)	4,284,900,000,000,000,000 (75 years)	$1.06 \times 10^{623,132,091}$

Key question: If we just want two buckets, where do we draw the dividing line?

# A Decidable Problem

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

# For Reference

- Assume  $c = 1$ .

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

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

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

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

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

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

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

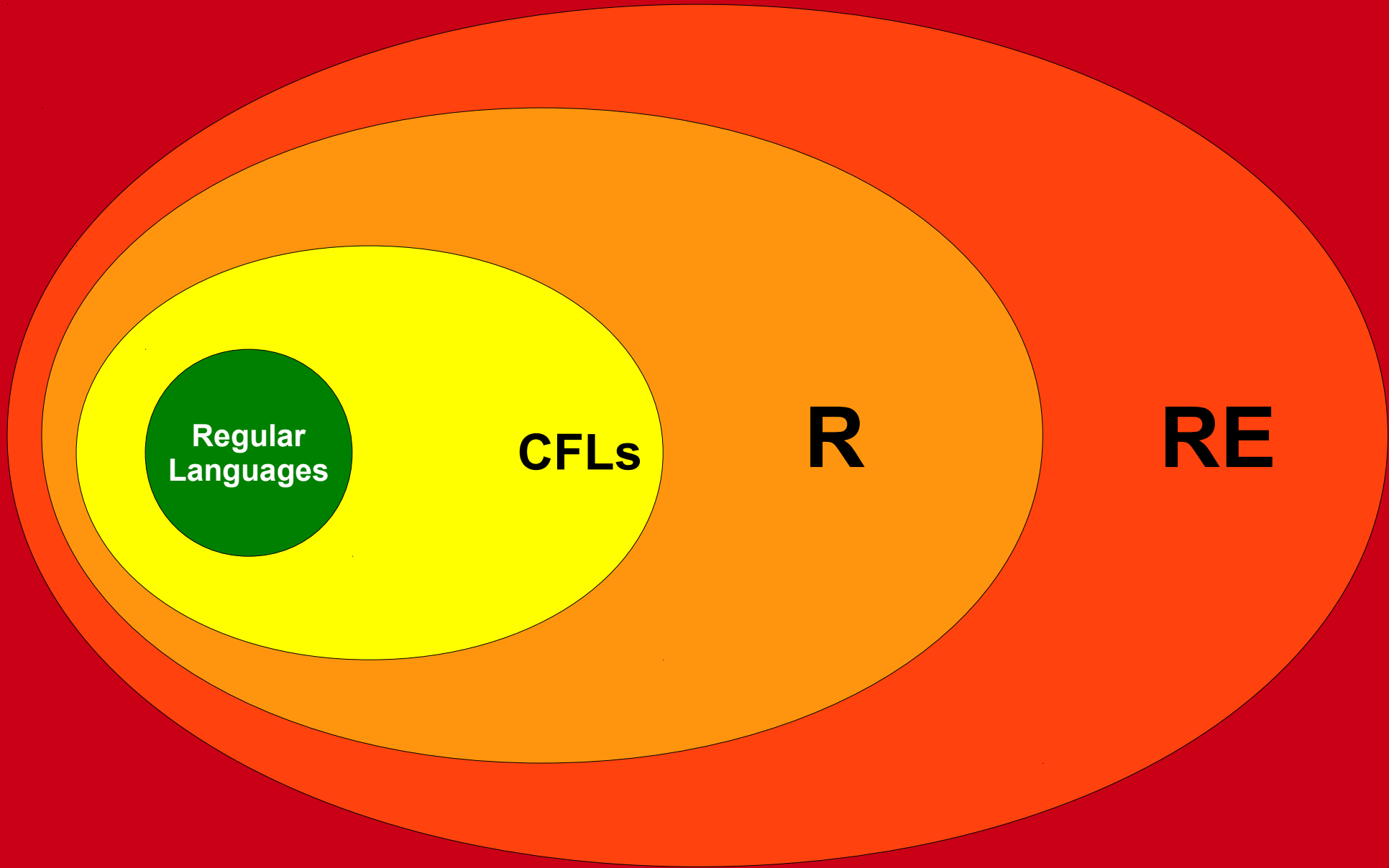
- (this is 500+ years)
- $n = 7$  would take time longer than the age of the universe

# 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.



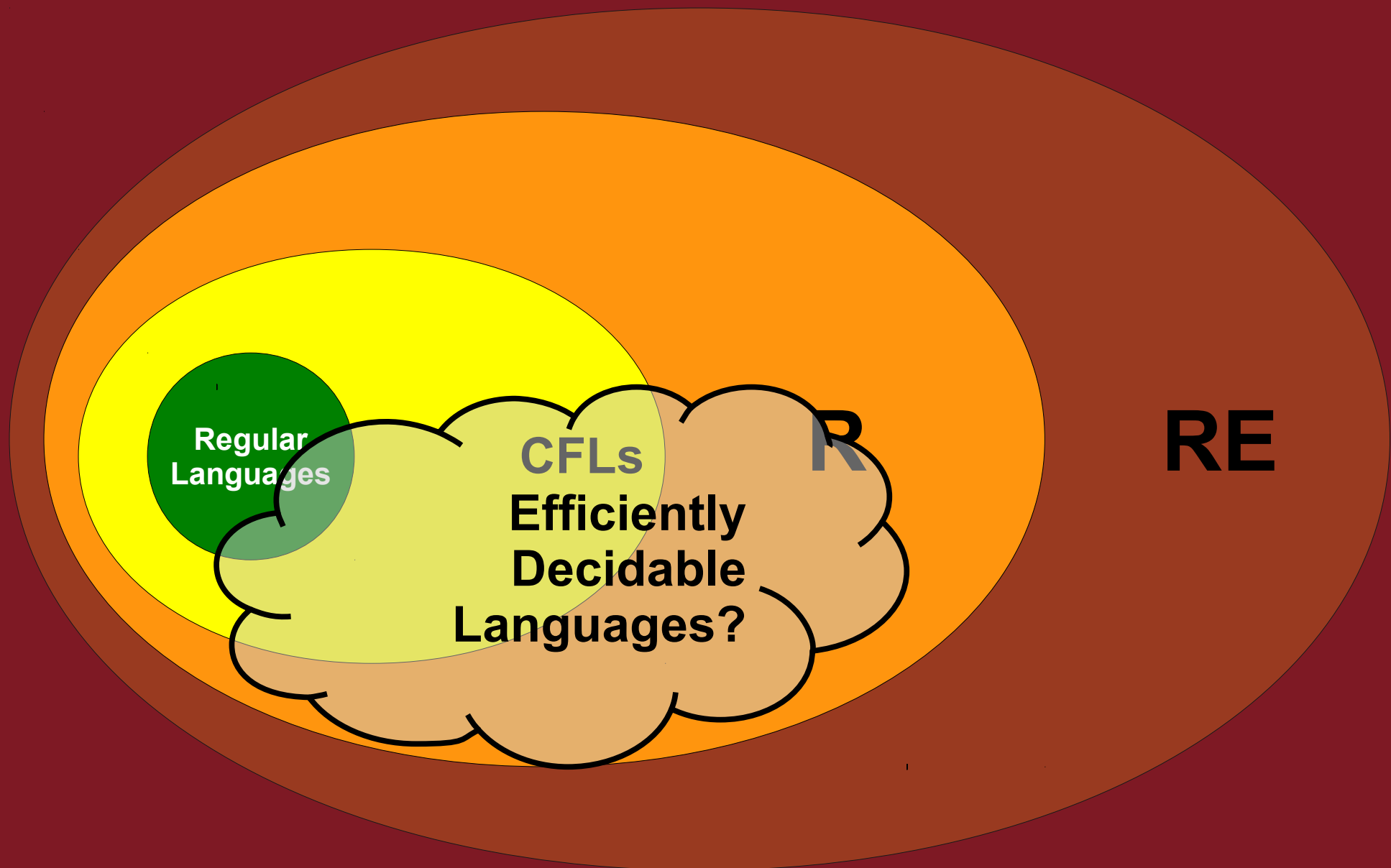
Regular  
Languages

CFLs

R

RE

All Languages



Regular Languages

CFLs  
Efficiently Decidable Languages?

R

RE

All Languages

# Measuring Complexity

- Suppose that we have a decider  $D$  for some language  $L$ .
- How might we measure the complexity of  $D$ ?

# Measuring Complexity

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

# Measuring Complexity

- Suppose that we have a decider  $D$  for some language  $L$ .
- How might we measure the complexity of  $D$ ?

Number of states.

Size of tape alphabet.

Size of input alphabet.

Amount of tape required.

- **Amount of time required.**

Number of times a given state is entered.

Number of times a given symbol is printed.

Number of times a given transition is taken.

(Plus a whole lot more...)

What is an efficient algorithm?

# Searching Finite Spaces

- Many decidable problems can be solved by searching over a large but finite space of possible options.
- Searching this space might take a staggeringly long time, but only finite time.
  - Recommended reading: *A Short Stay in Hell*, by Steven Peck
- From a decidability perspective, this is totally fine!
- From a complexity perspective, this may be totally unacceptable.

# Defining Efficiency

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

# Polynomials and Exponentials

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

# The Cobham-Edmonds Thesis

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

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

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

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

# The Cobham-Edmonds Thesis

According to the Cobham-Edmonds thesis, how many of the following runtimes are considered efficient?

$$4n^2 - 3n + 137$$

$$10^{500}$$

$$2^n$$

$$1.00000000000001^n$$

$$n^{1,000,000,000,000}$$

$$n^{\log n}$$

# The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 - 2n^2 + 4n$
  - $n \log \log n$
- “Efficient” runtimes:
  - $n^{1,000,000,000,000}$
  - $10^{500}$
- Inefficient runtimes:
  - $2^n$
  - $n!$
  - $n^n$
- “Inefficient” runtimes:
  - $n^{0.0001 \log n}$
  - $1.0000000001^n$

# Why Polynomials?

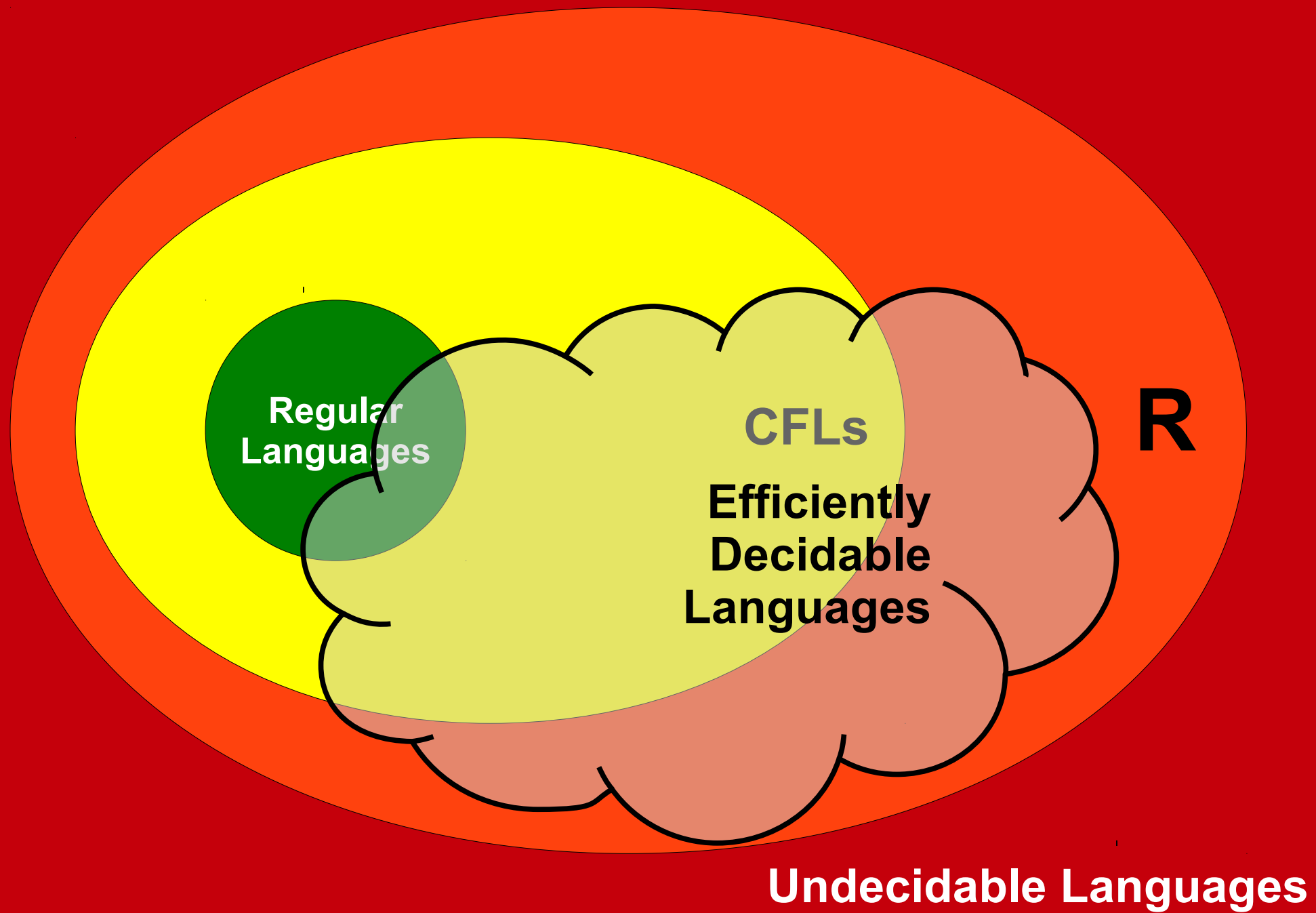
- Polynomial time *somewhat* captures efficient computation, but has a few edge cases.
- However, polynomials have very nice mathematical properties:
  - The sum of two polynomials is a polynomial. (Running one efficient algorithm, then another, gives an efficient algorithm.)
  - The product of two polynomials 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.)

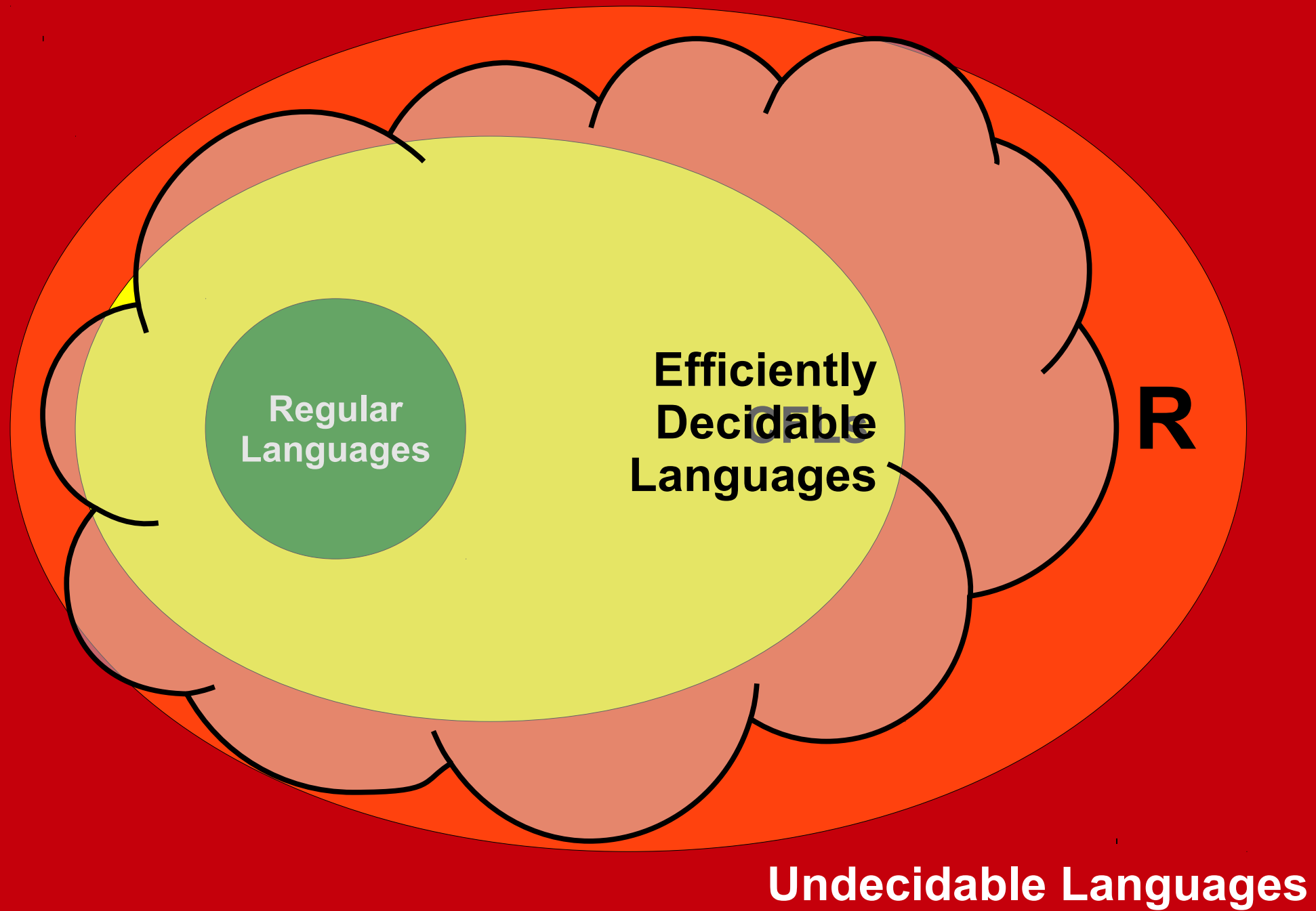
# The Complexity Class **P**

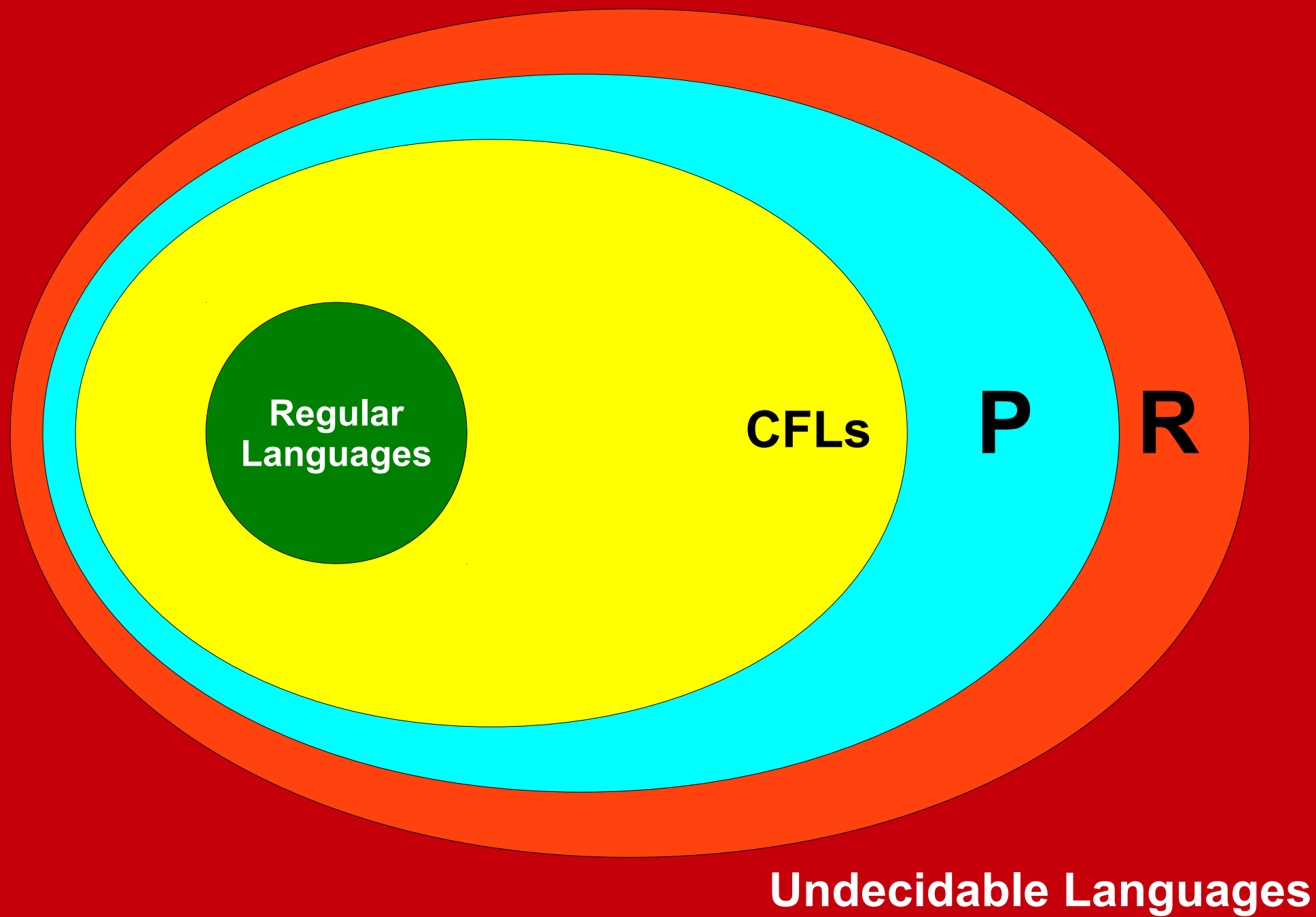
- The ***complexity class P*** (for *p*olynomial time) contains all problems that can be solved in polynomial time.
- Formally:  
$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$
- Assuming the Cobham-Edmonds thesis, a language is in **P** if it can be decided efficiently.

# Examples of Problems in **P**

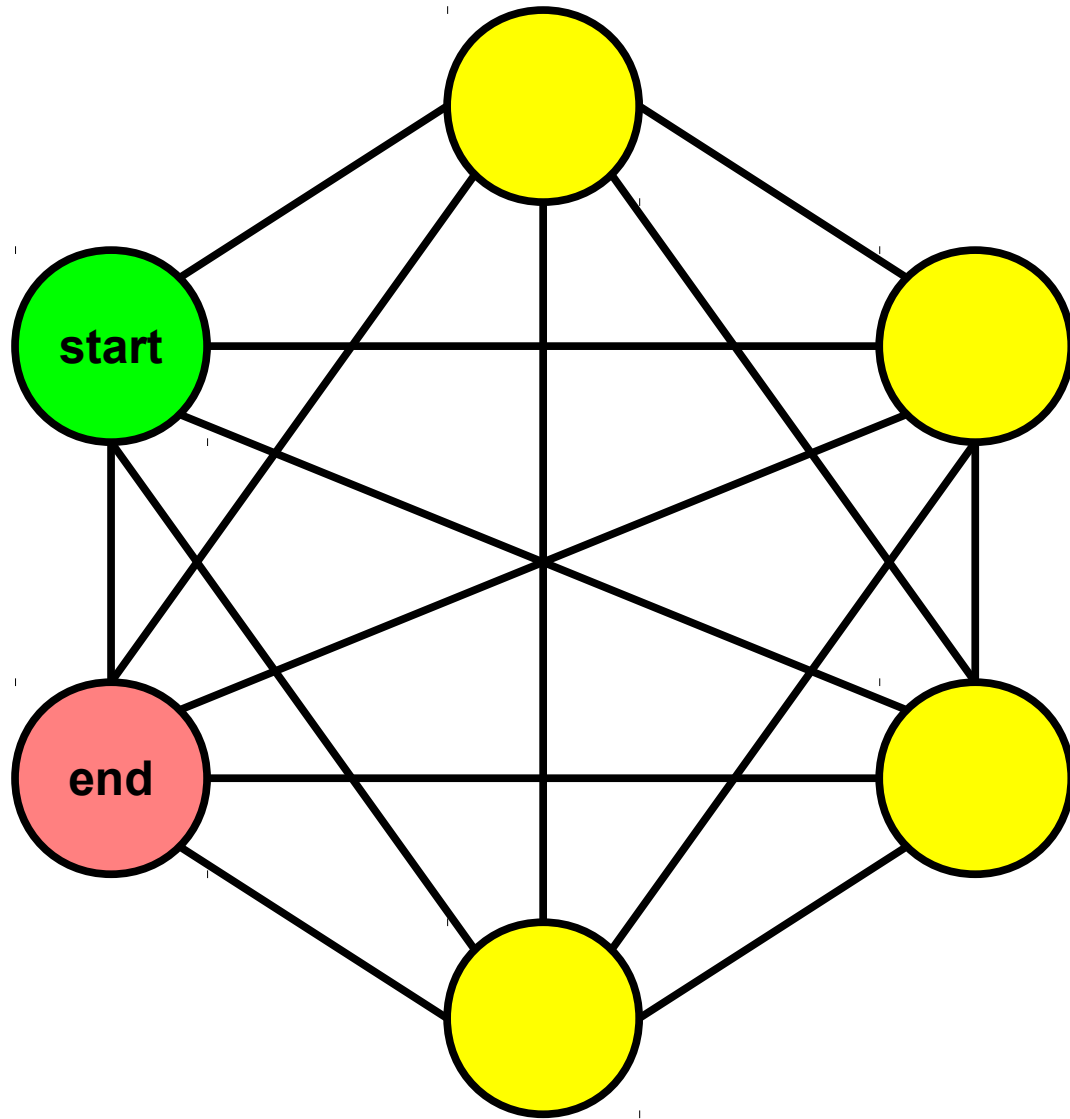
- All regular languages are in **P**.
  - All have linear-time TMs.
- All CFLs are in **P**.
  - Requires a more nuanced argument (the *CYK algorithm* or *Earley's algorithm*)
- And a *ton* of other problems are in **P** as well.
  - Curious? Take CS161!







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



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



List all the subsets  
of a given set.

# 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

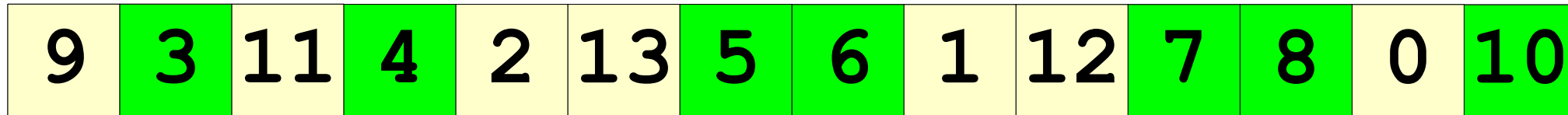
Does this Sudoku problem  
have a solution?

# Verifiers - Again

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

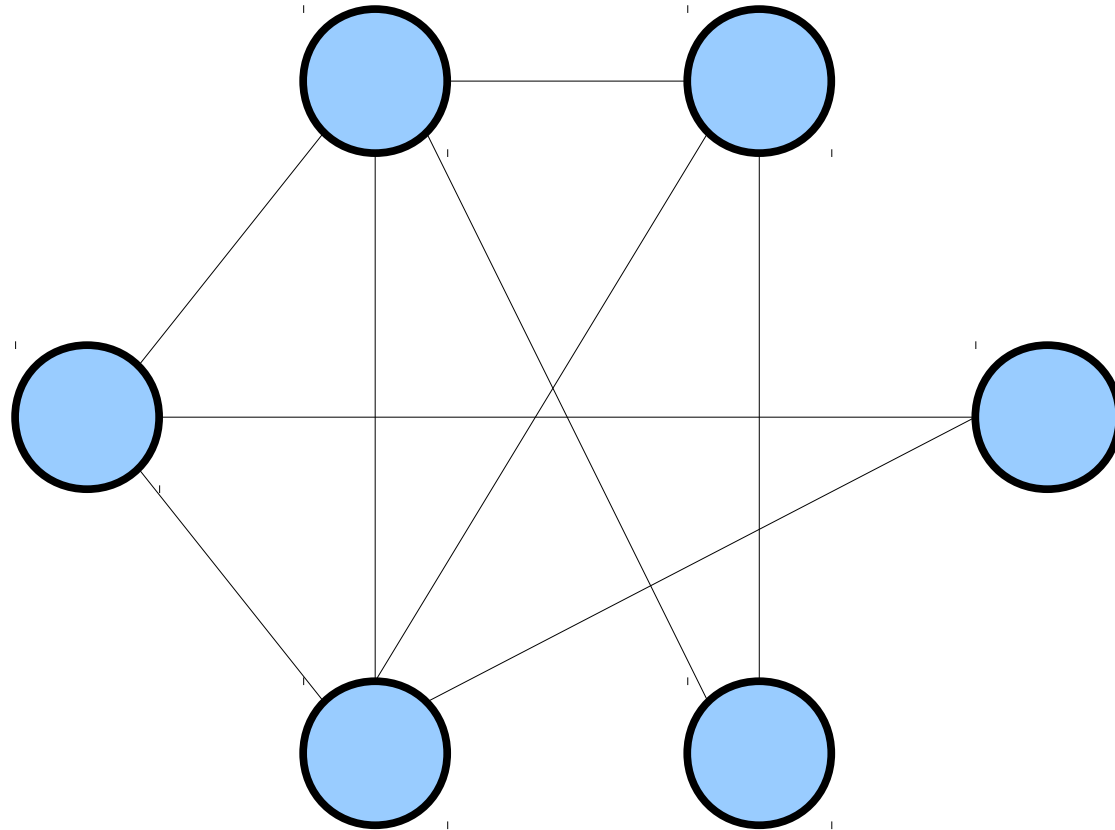
Is there an ascending subsequence of  
length at least 7?

# Verifiers - Again



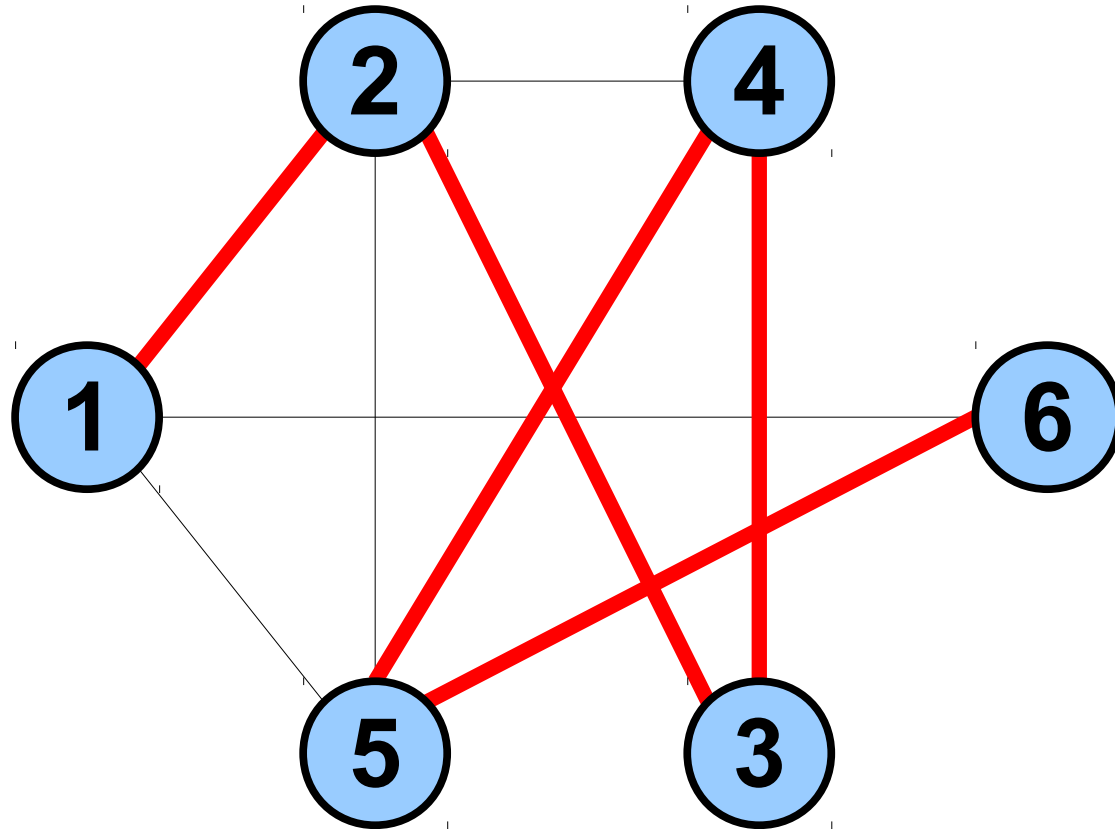
Is there an ascending subsequence of  
length at least 7?

# Verifiers - Again



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

# Verifiers - Again



Is there a simple 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.
  - $w \in L$  iff  $\exists c \in \Sigma^*. V$  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.
  - $w \in L$  iff  $\exists c \in \Sigma^*. V$  accepts  $\langle w, c \rangle$ .
  - $V$ 's runtime is a polynomial in  $|w|$  (that is,  $V$ 's runtime is  $O(|w|^k)$  for some integer  $k$ )

# The Complexity Class **NP**

- The complexity class **NP** (*nondeterministic polynomial time*) contains all problems that can be verified in polynomial time.
- Formally:
$$\mathbf{NP} = \{ L \mid \text{There is a polynomial-time verifier for } L \}$$
- 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.
- Although it’s not immediately obvious, **NP**  $\subseteq$  **R**. Come talk to me after class if you’re curious why!

And now...

The

***Most Important Question***

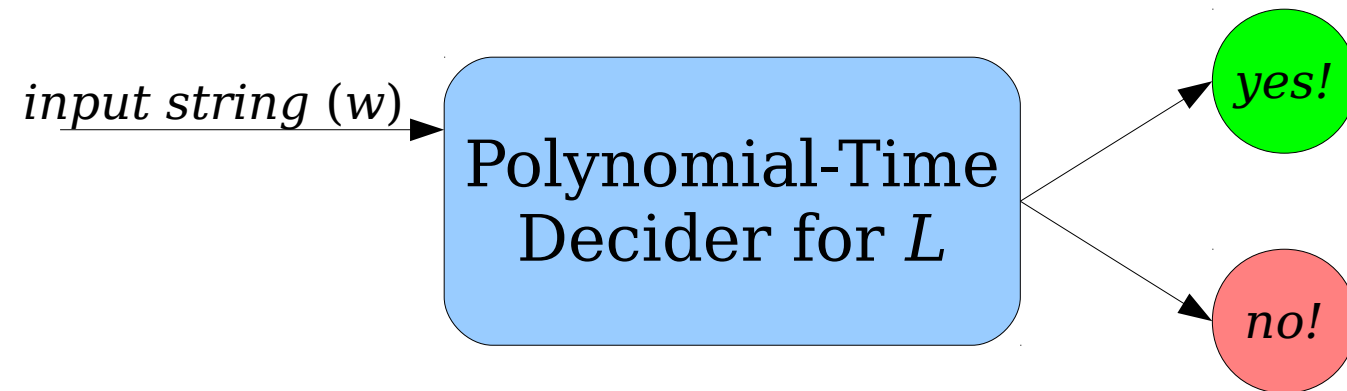
in

***Theoretical Computer Science***

What is the connection between **P** and **NP**?

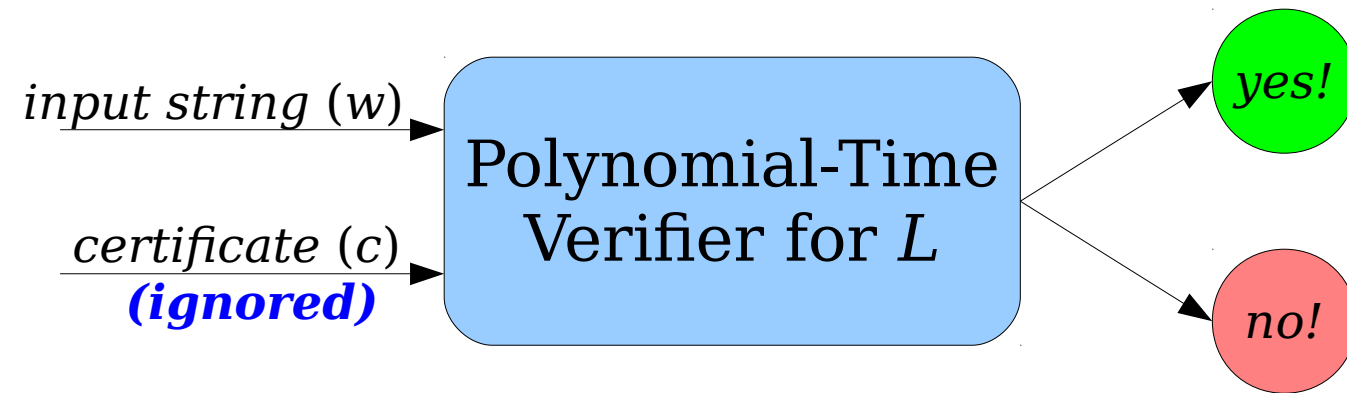
**P** = {  $L$  | There is a polynomial-time decider for  $L$  }

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



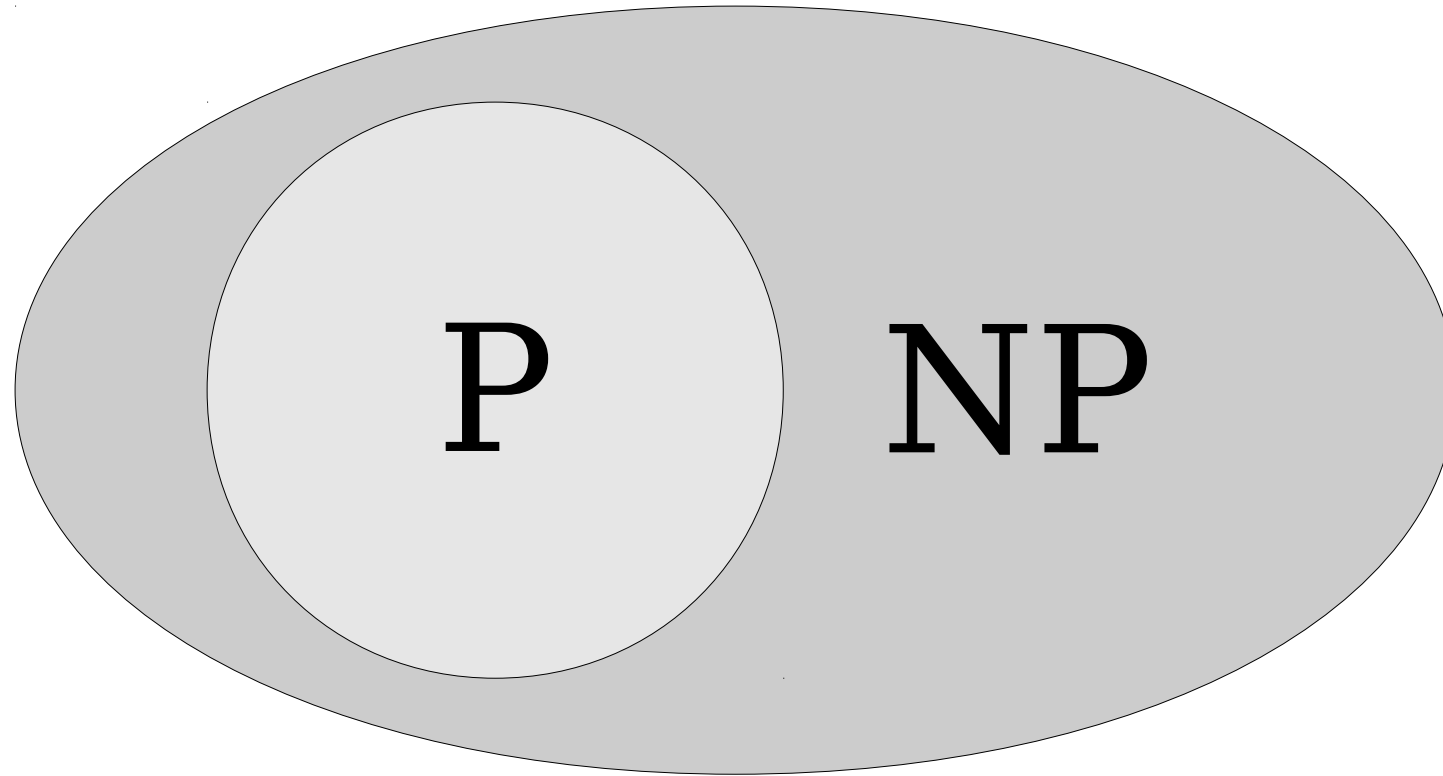
**P** = {  $L$  | There is a polynomial-time decider for  $L$  }

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

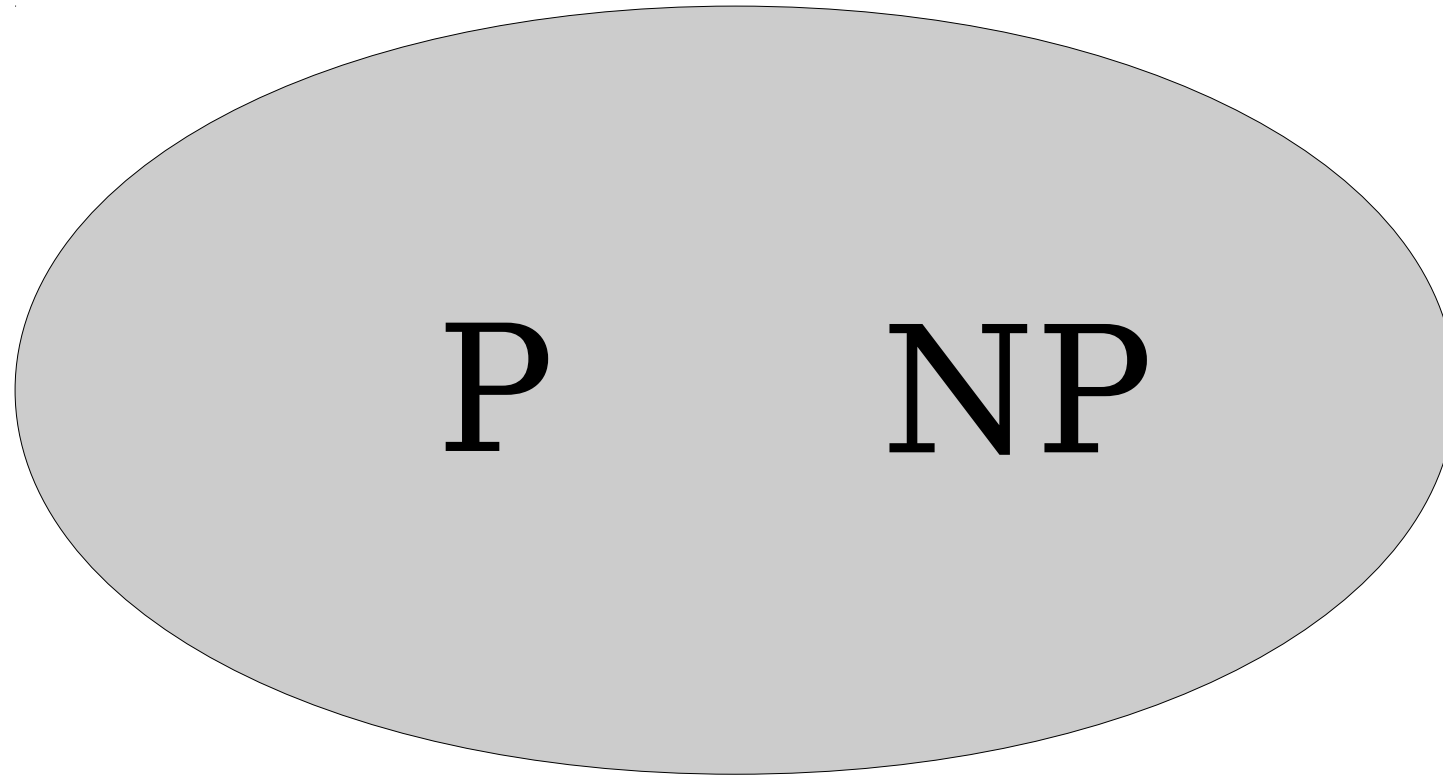


**P**  $\subseteq$  **NP**

Which Picture is Correct?



Which Picture is Correct?



Does **P** = **NP**?

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

- The  $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$  question is the most important question in theoretical computer science.
- With the verifier definition of  $\mathbf{NP}$ , one way of phrasing this question is  
*If a solution to a problem can be **checked** efficiently, can that problem 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  $P \neq NP$ , *none* of these problems have efficient solutions.

# Why This Matters

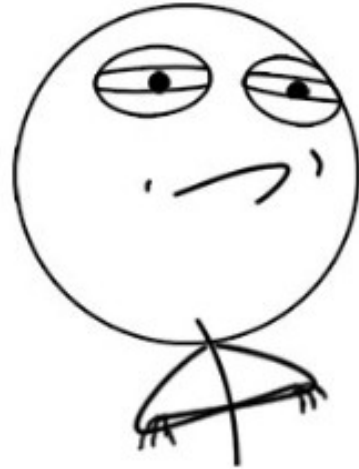
- If **P = NP**:
  - A huge number of seemingly difficult problems could be solved efficiently.
  - Our capacity to solve many problems will scale well with the size of the problems we want to solve.
- If **P  $\neq$  NP**:
  - Enormous computational power would be required to solve many seemingly easy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

# What We Know

- Resolving  $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$  has proven *extremely difficult*.
- In the past 45 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  $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ .
  - A majority of computer scientists believe  $\mathbf{P} \neq \mathbf{NP}$ , but this isn't a large majority.
- Interesting read: Interviews with leading thinkers about  $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ :
  - <http://web.eng.puc.cl/~jabaier/iic2212/poll-1.pdf>

# The Million-Dollar Question

**CHALLENGE ACCEPTED**



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

Do you think **P** = **NP**?

What do we know about  $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ ?

# Adapting our Techniques

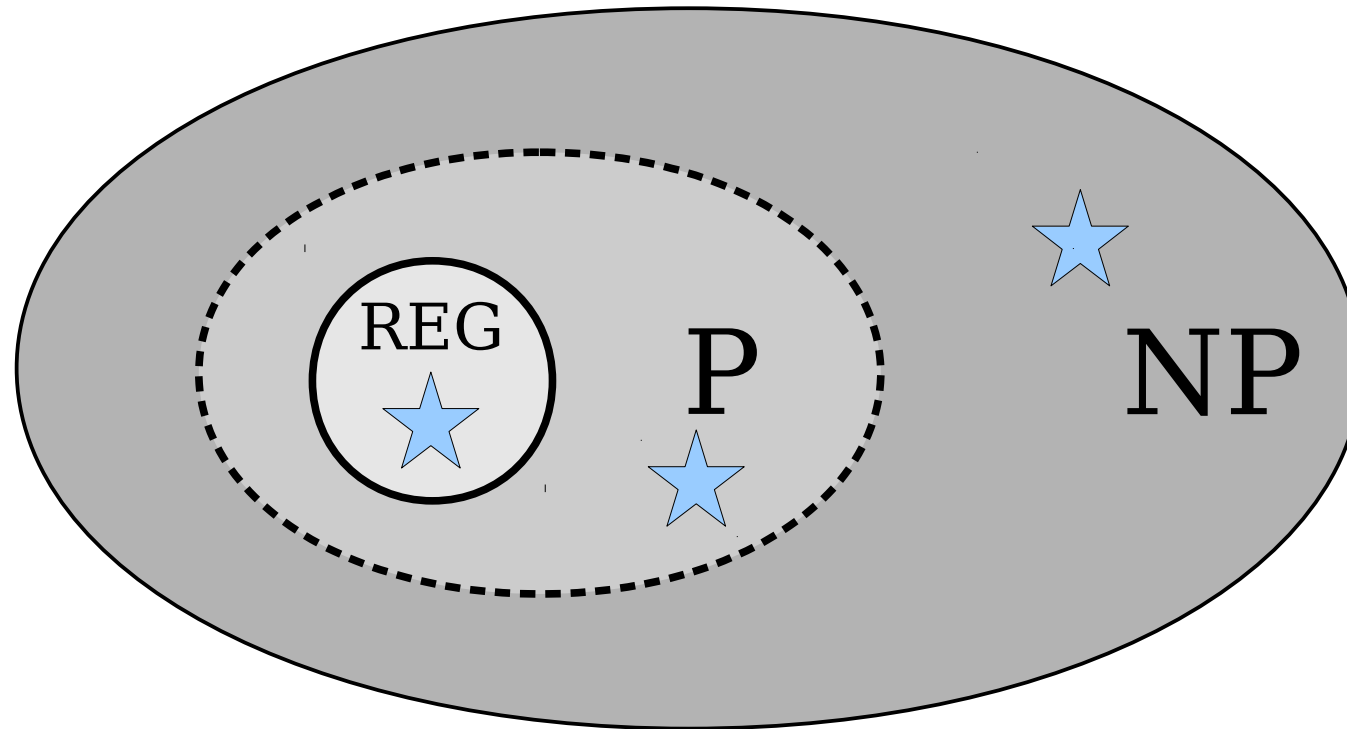
# A Problem

- 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 run other TMs as subroutines.
  - **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**?



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

How can we rank the relative difficulties of problems?

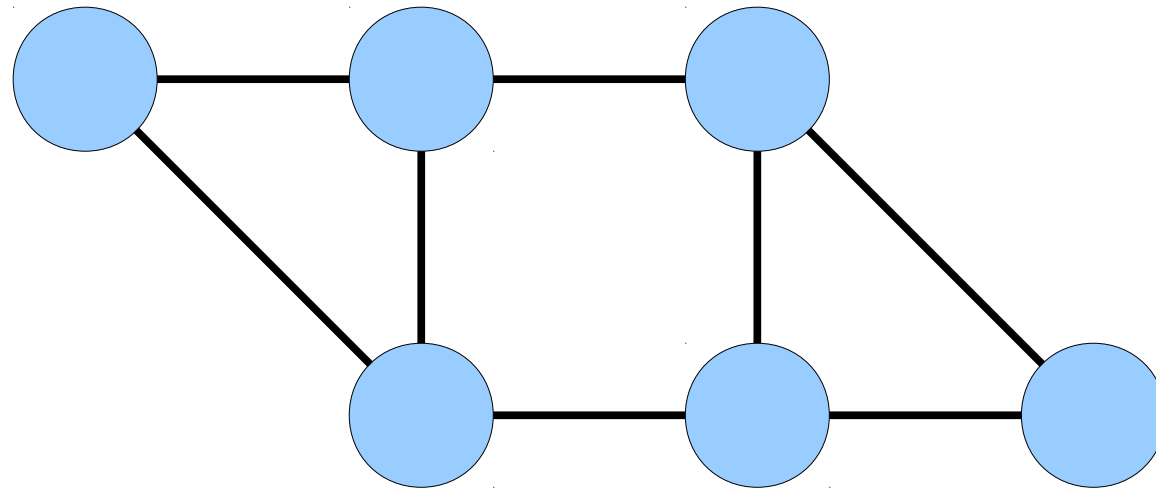
# Reducibility

# 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.

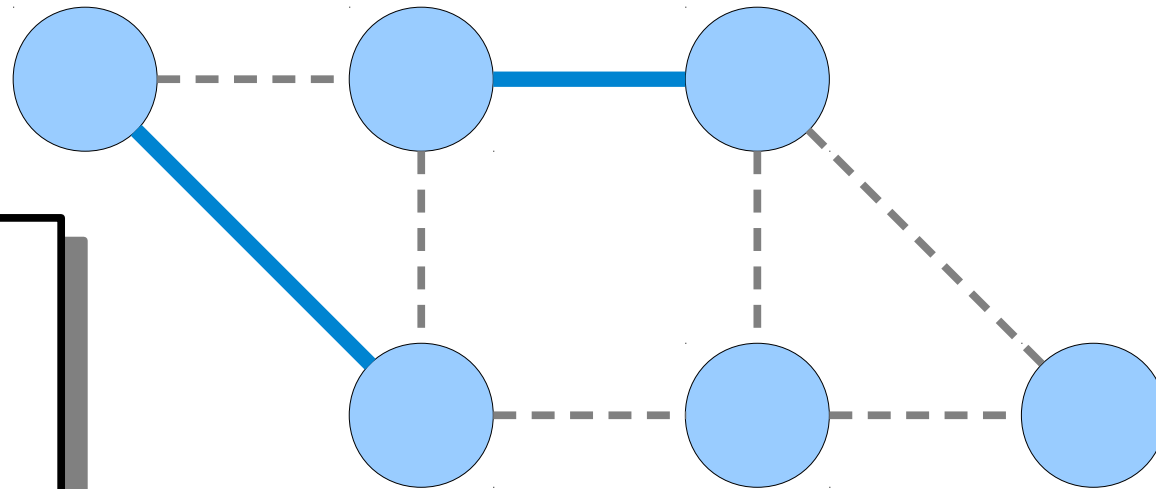
# Maximum Matching

- 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.



# Maximum Matching

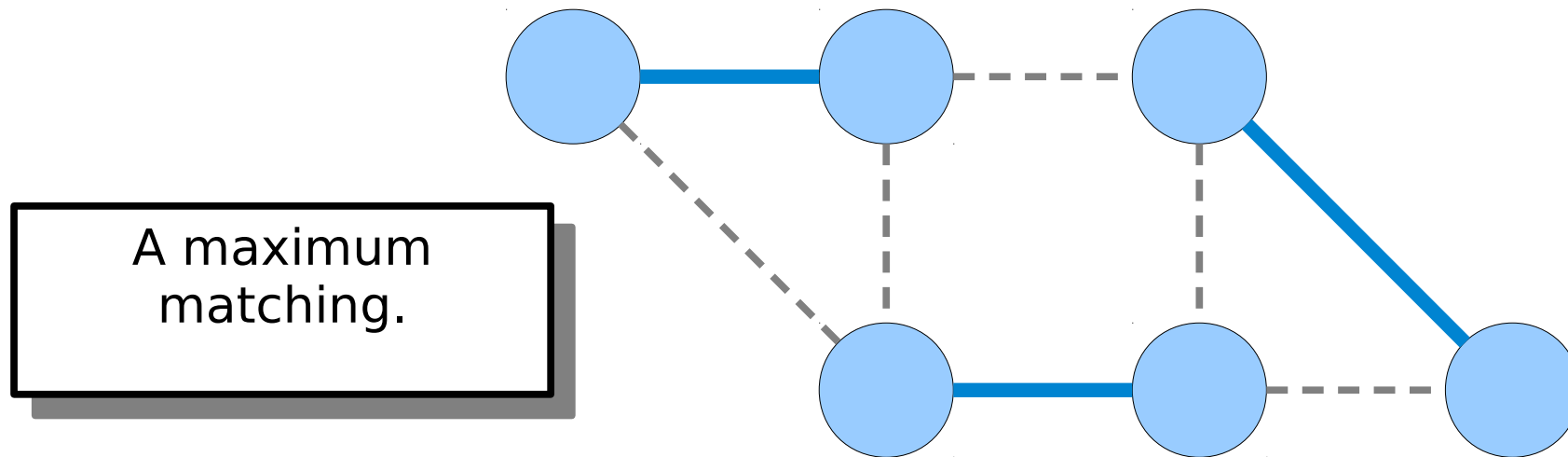
- Given an undirected graph  $G$ , a **matching** in  $G$  is a set of edges such that no two edges share an endpoint.
- A **maximum matching** is a matching with the largest number of edges.



A matching, but  
not a maximum  
matching.

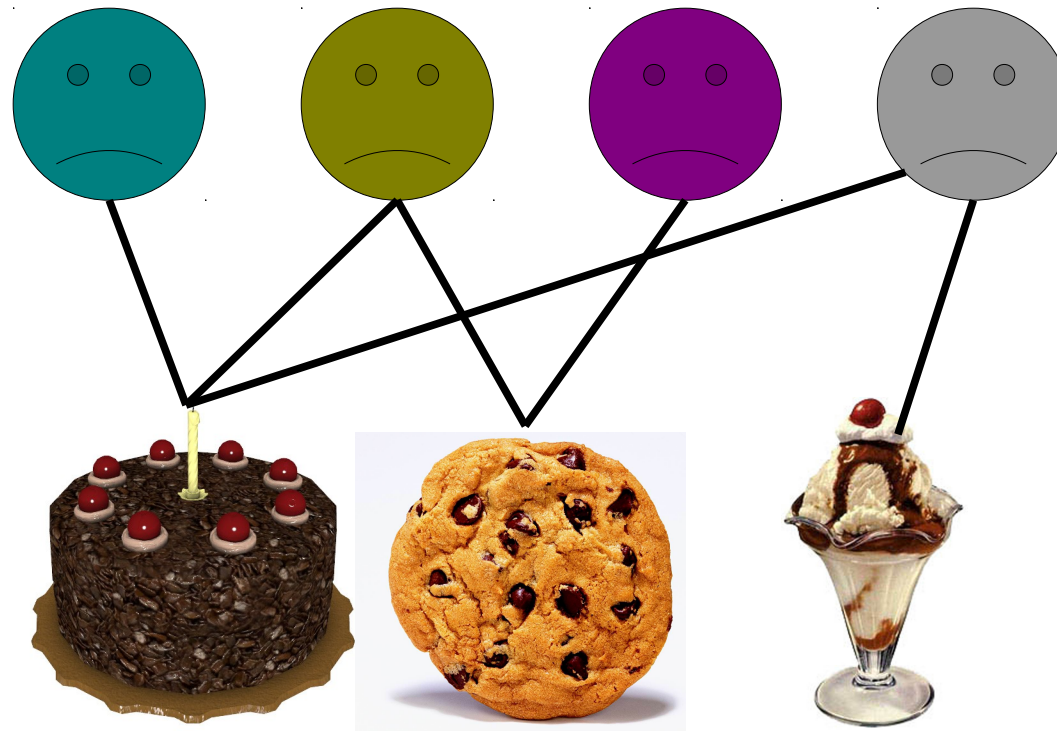
# Maximum Matching

- 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.



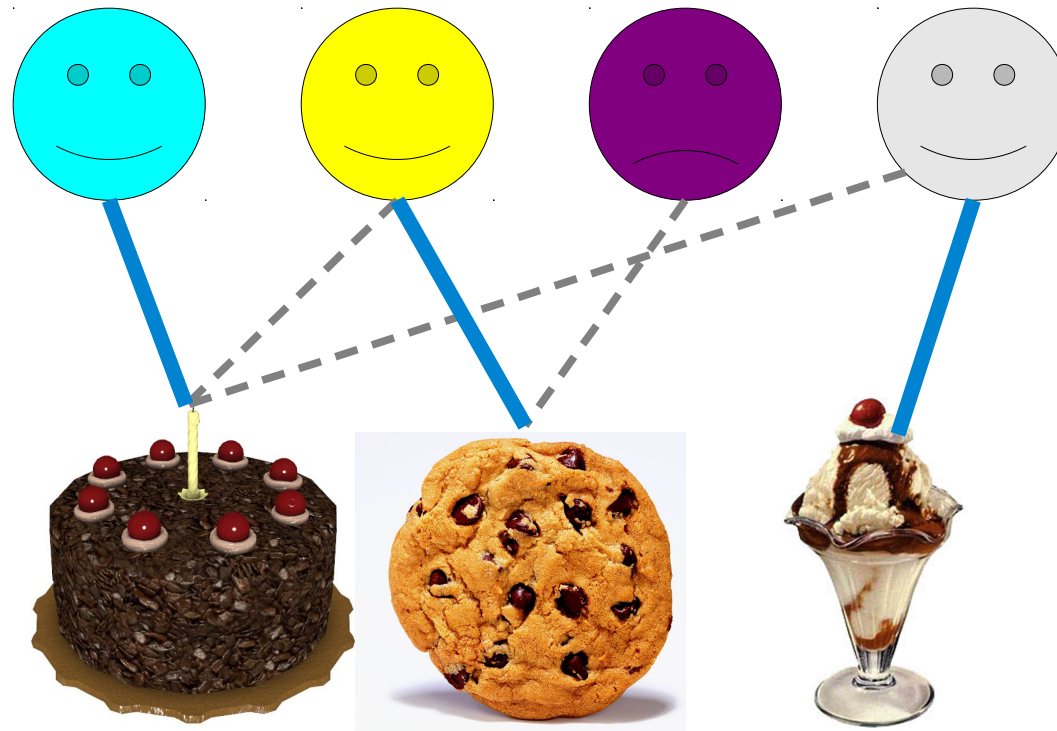
# Maximum Matching

- 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

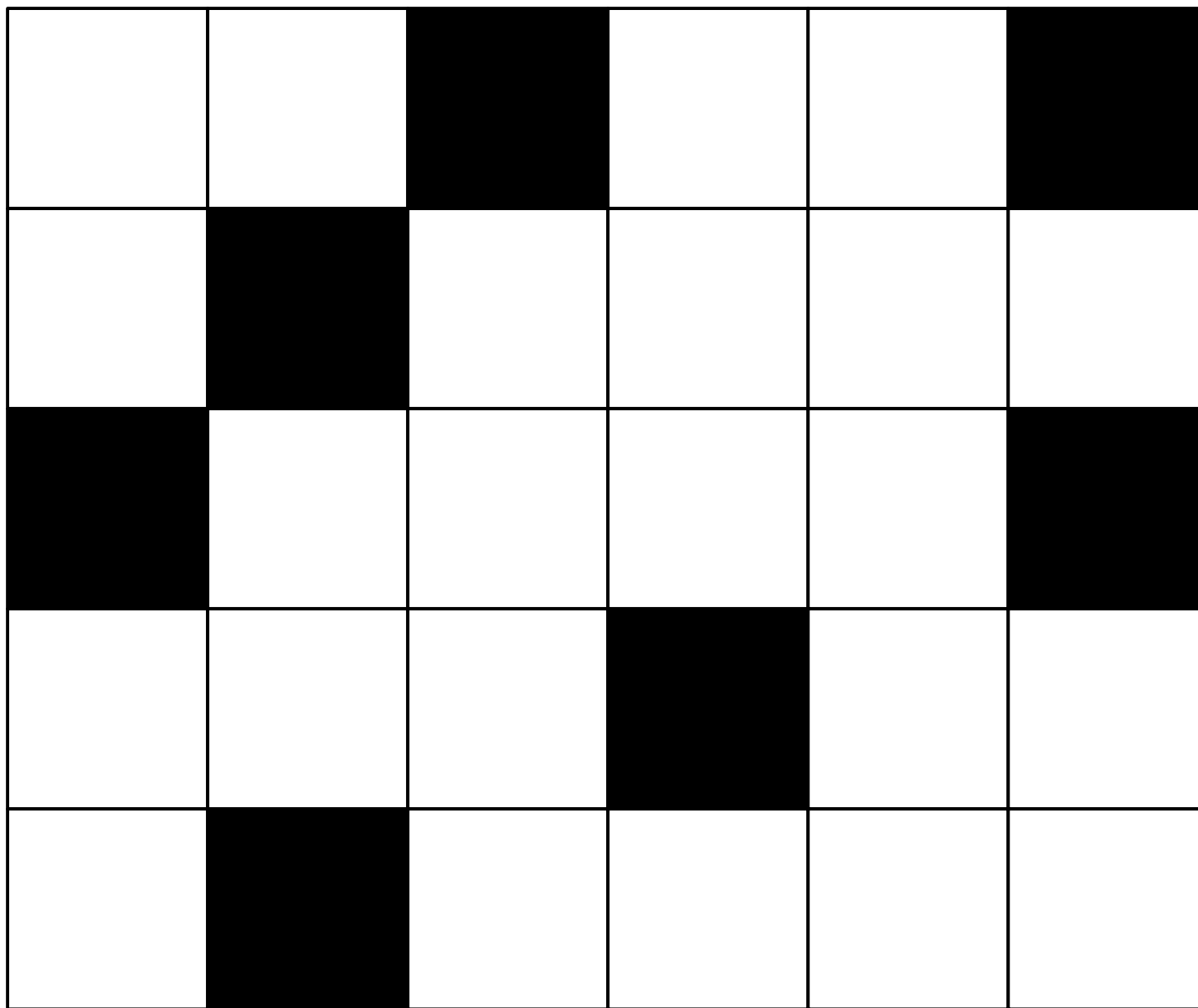
- Given an undirected graph  $G$ , a **matching** in  $G$  is a set of edges such that no two edges share an endpoint.
- A **maximum matching** is a matching with the largest number of edges.



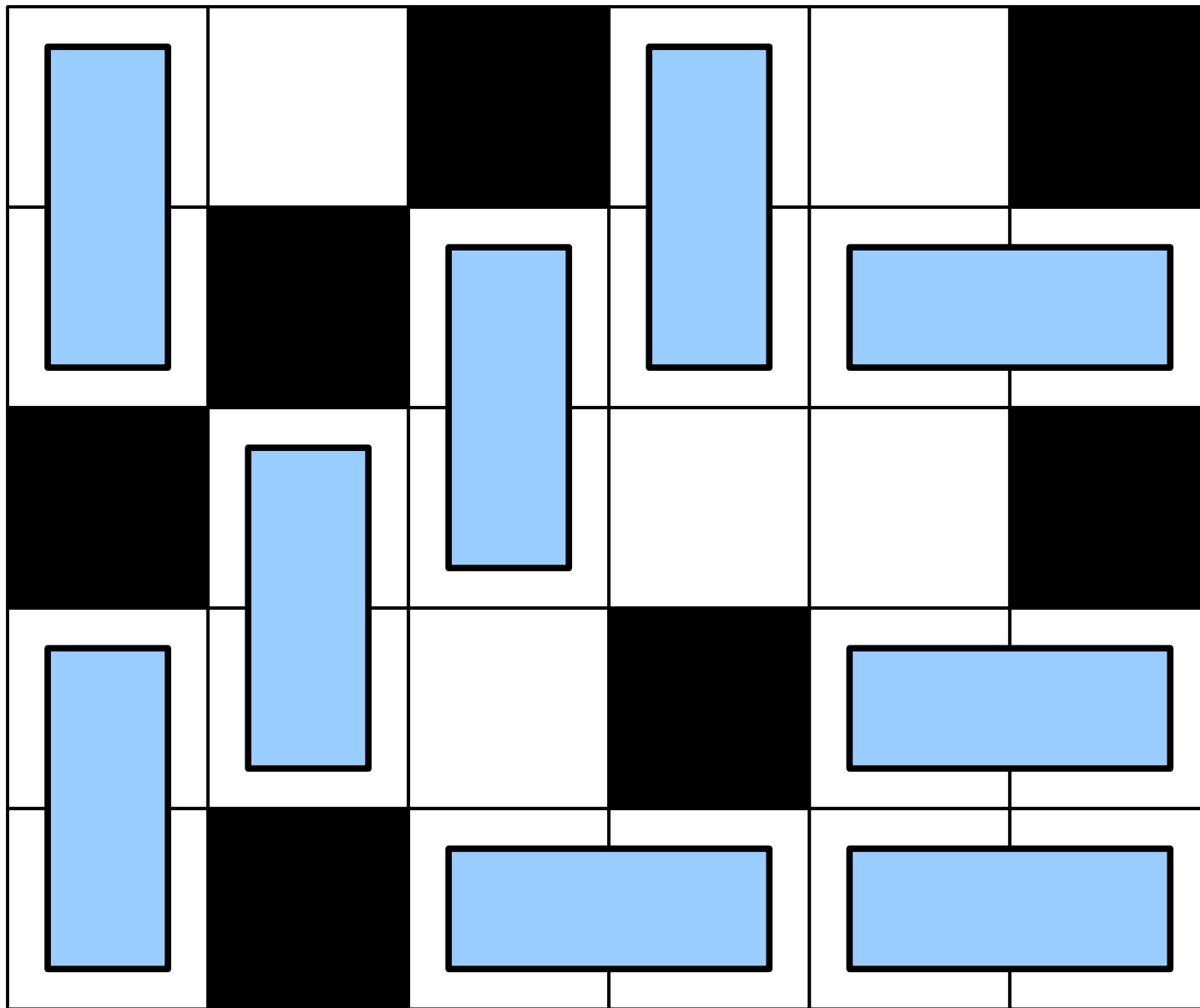
# Maximum Matching

- Jack Edmonds' paper “Paths, Trees, and Flowers” gives a polynomial-time algorithm for finding maximum matchings.
  - (This is the same Edmonds as in “Cobham-Edmonds Thesis.”)
- Using this fact, what other problems can we solve?

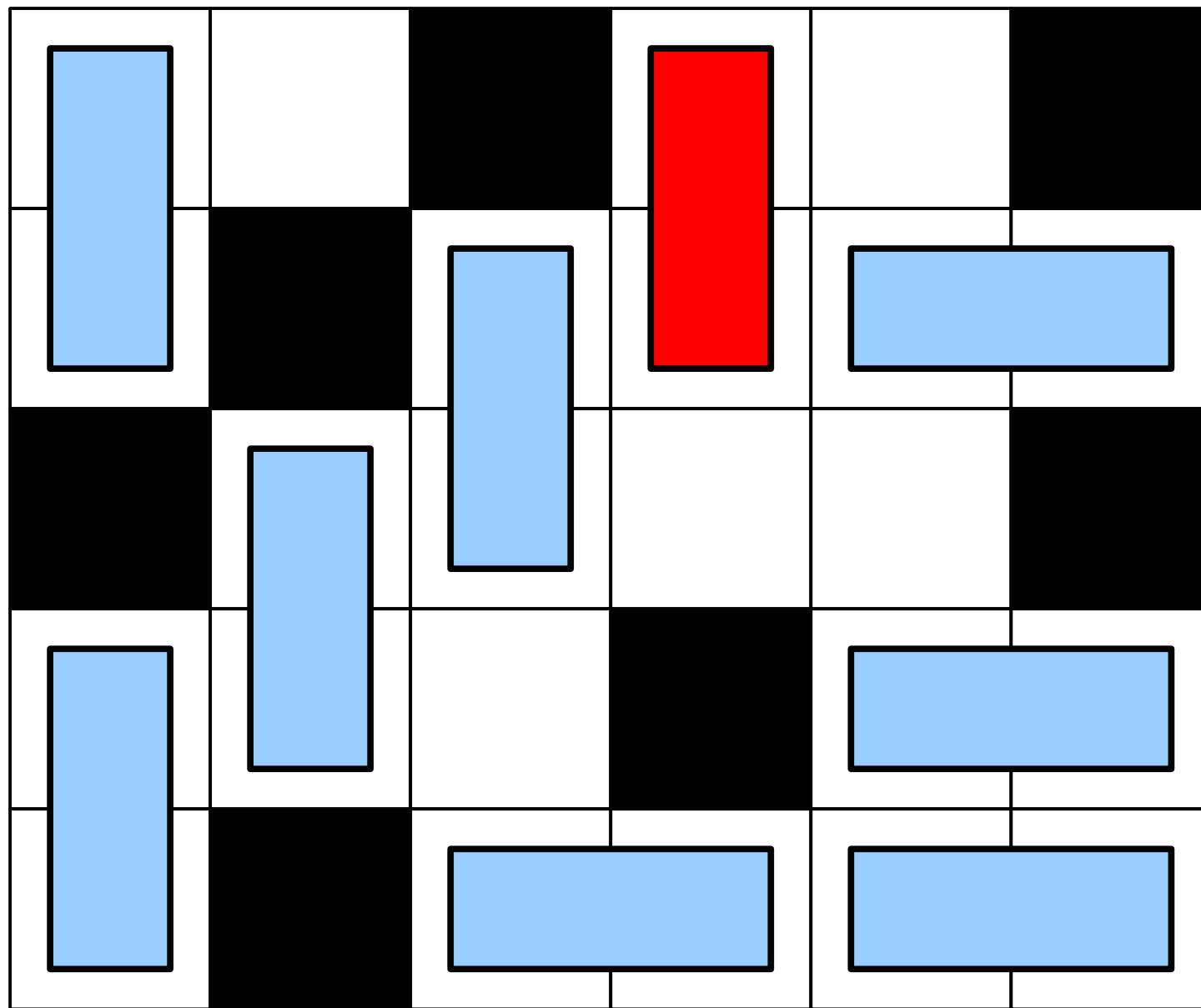
# Domino Tiling



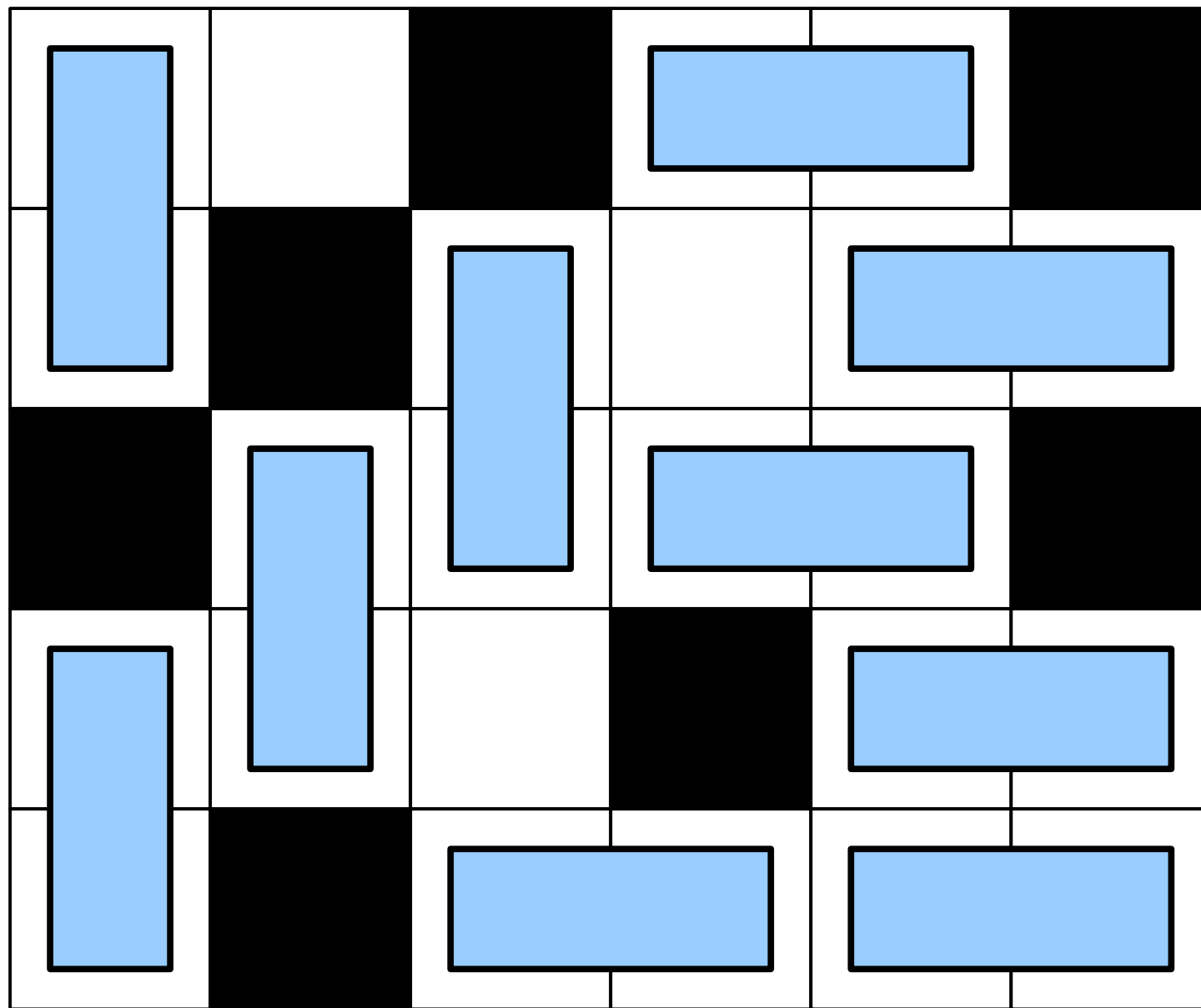
# Domino Tiling



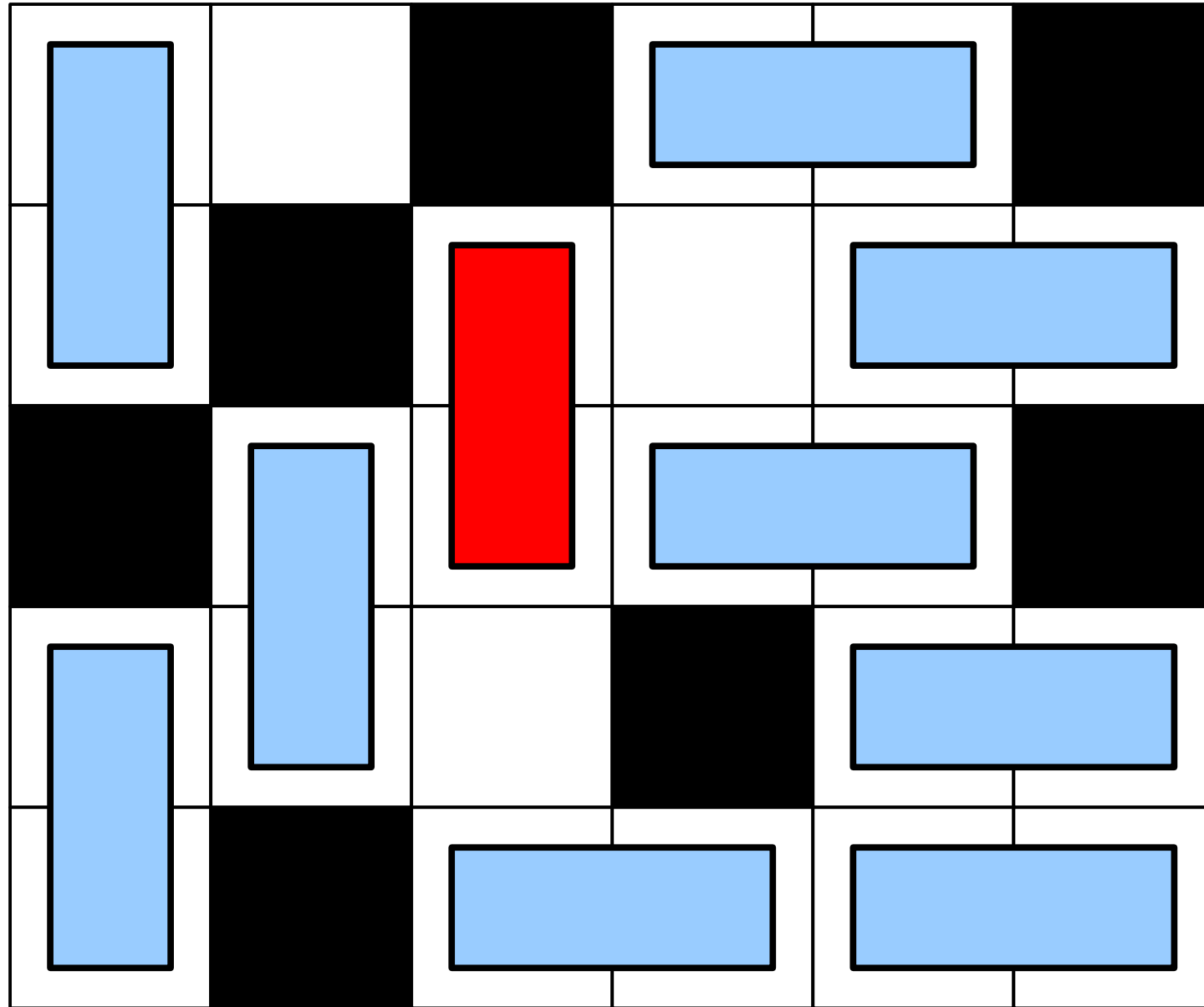
# Domino Tiling



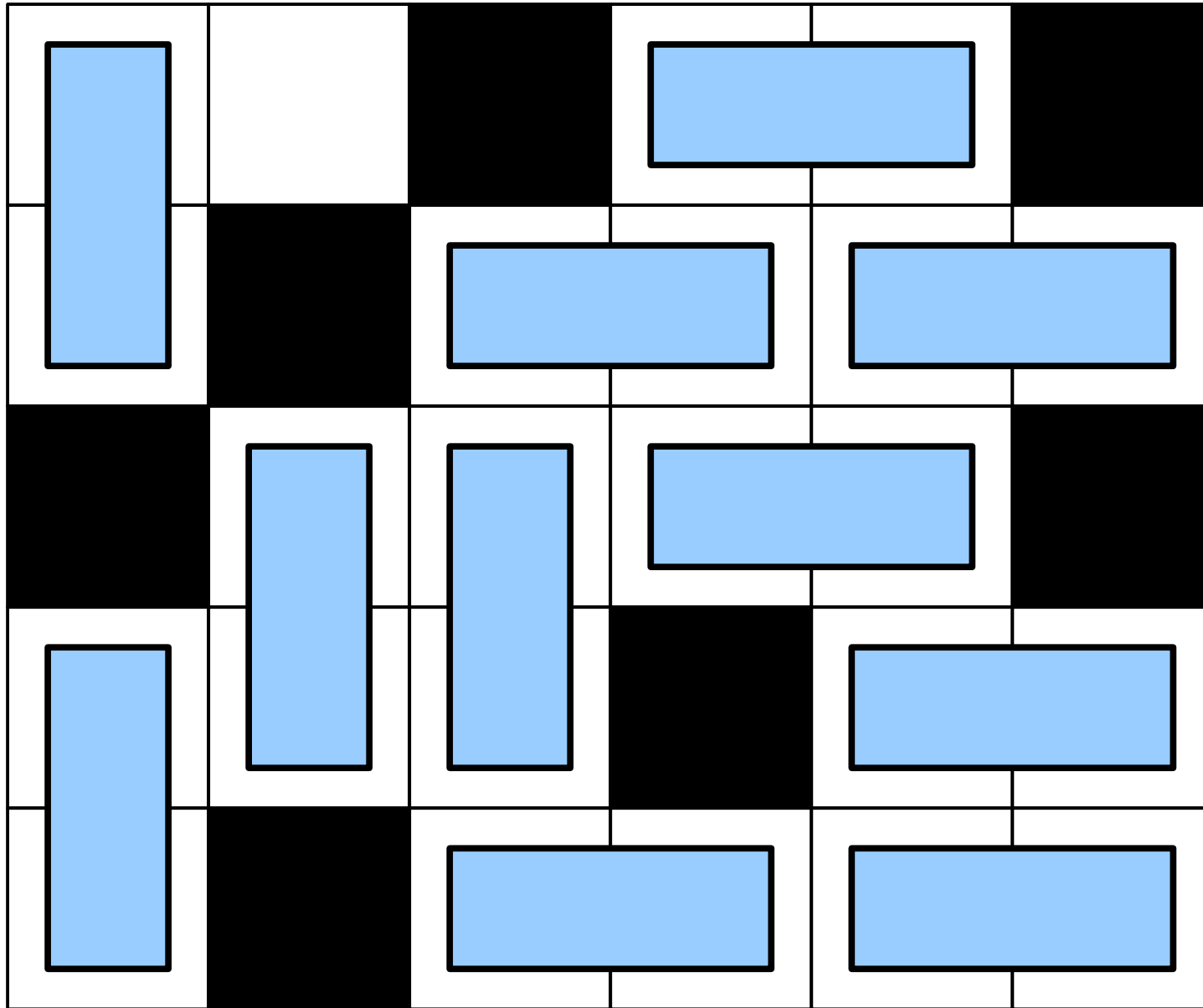
# Domino Tiling



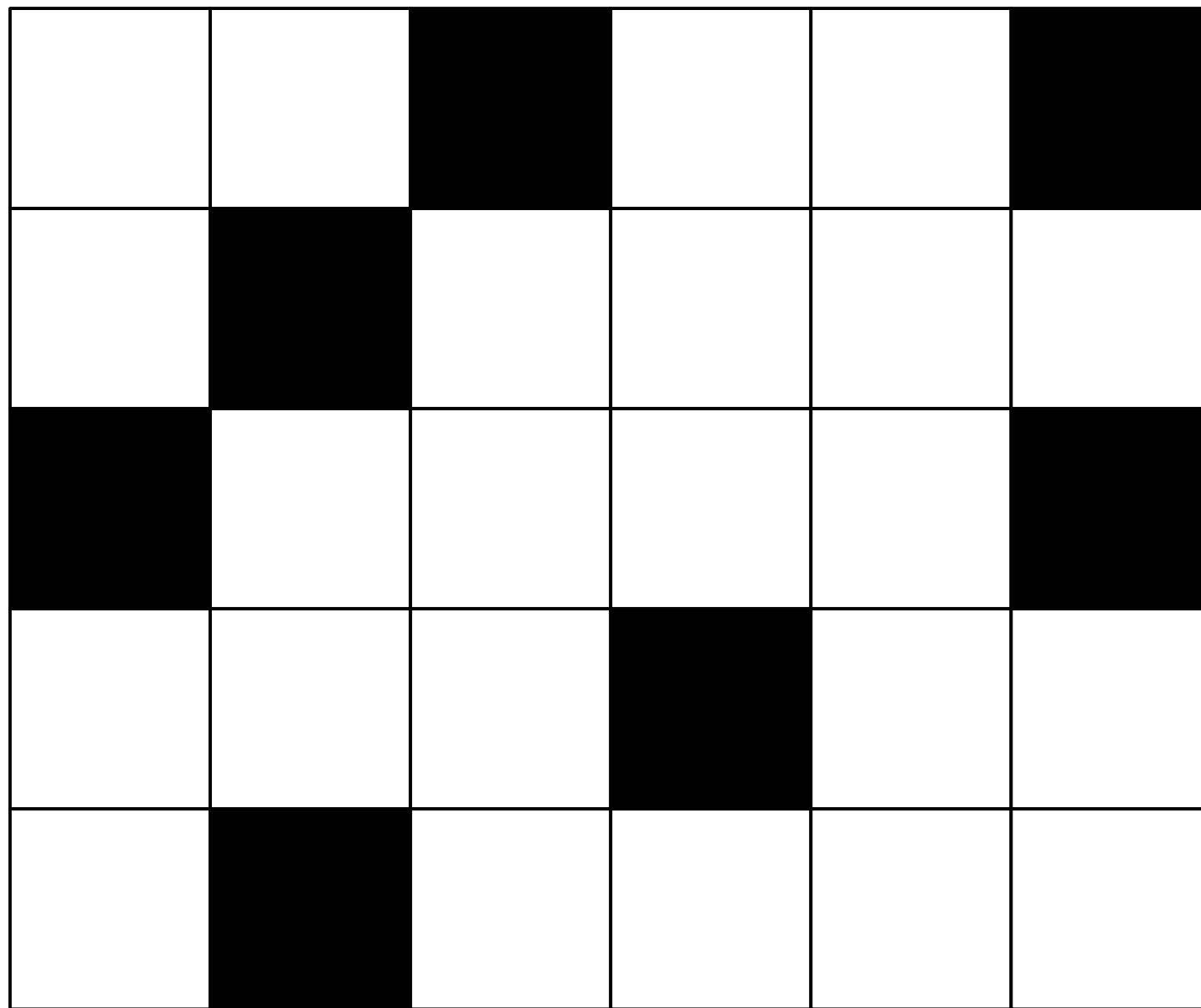
# Domino Tiling



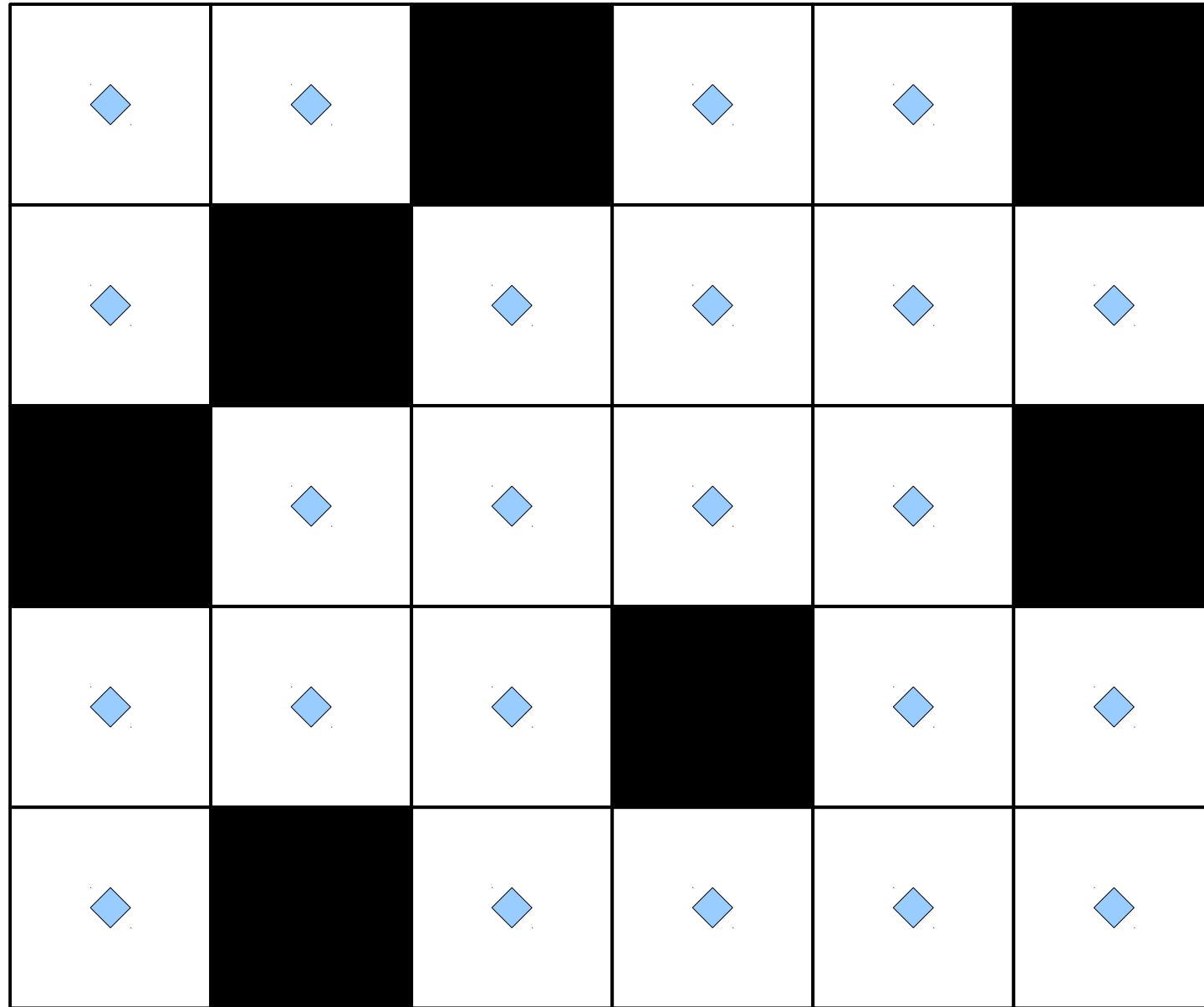
# Domino Tiling



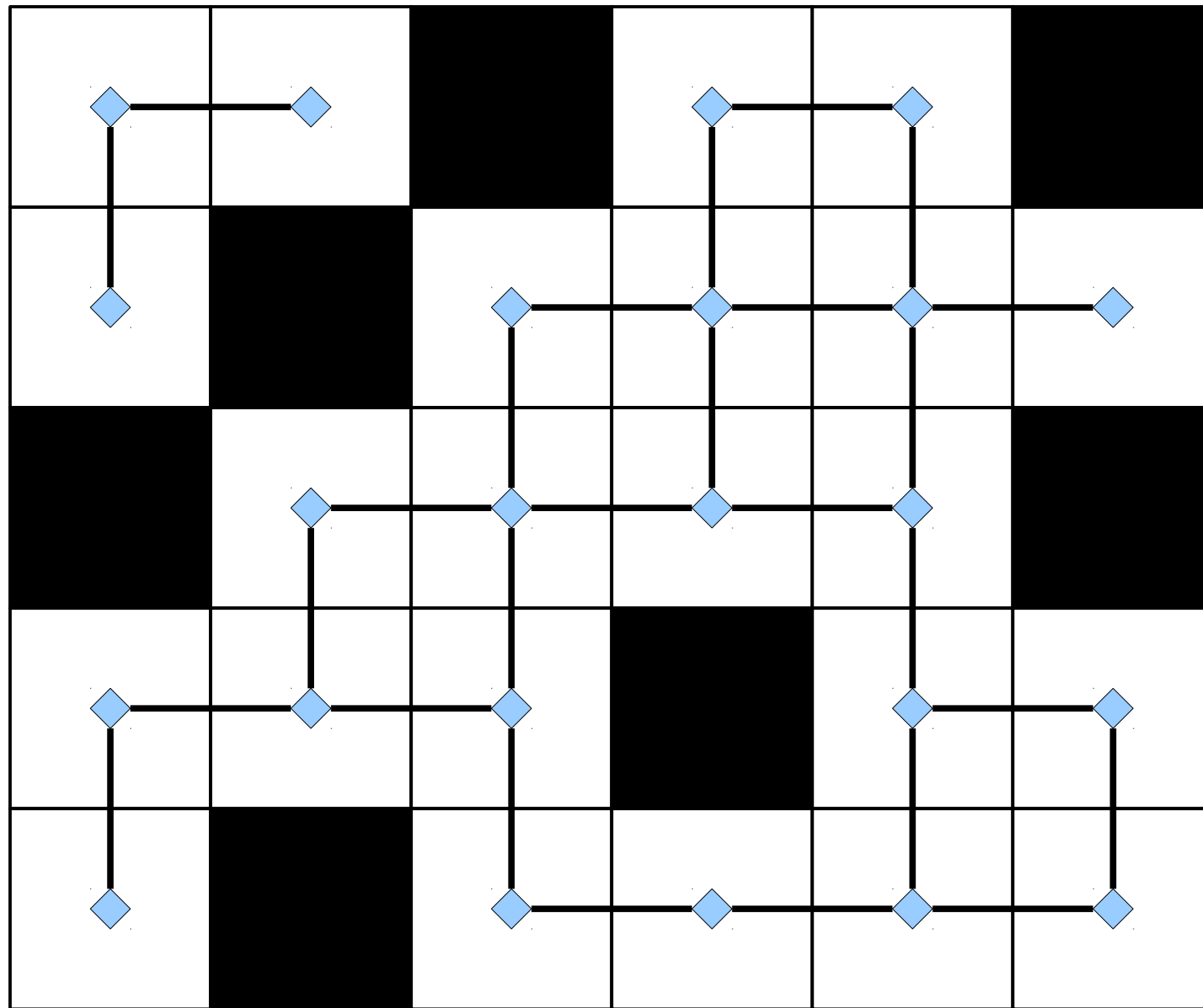
# Solving Domino Tiling



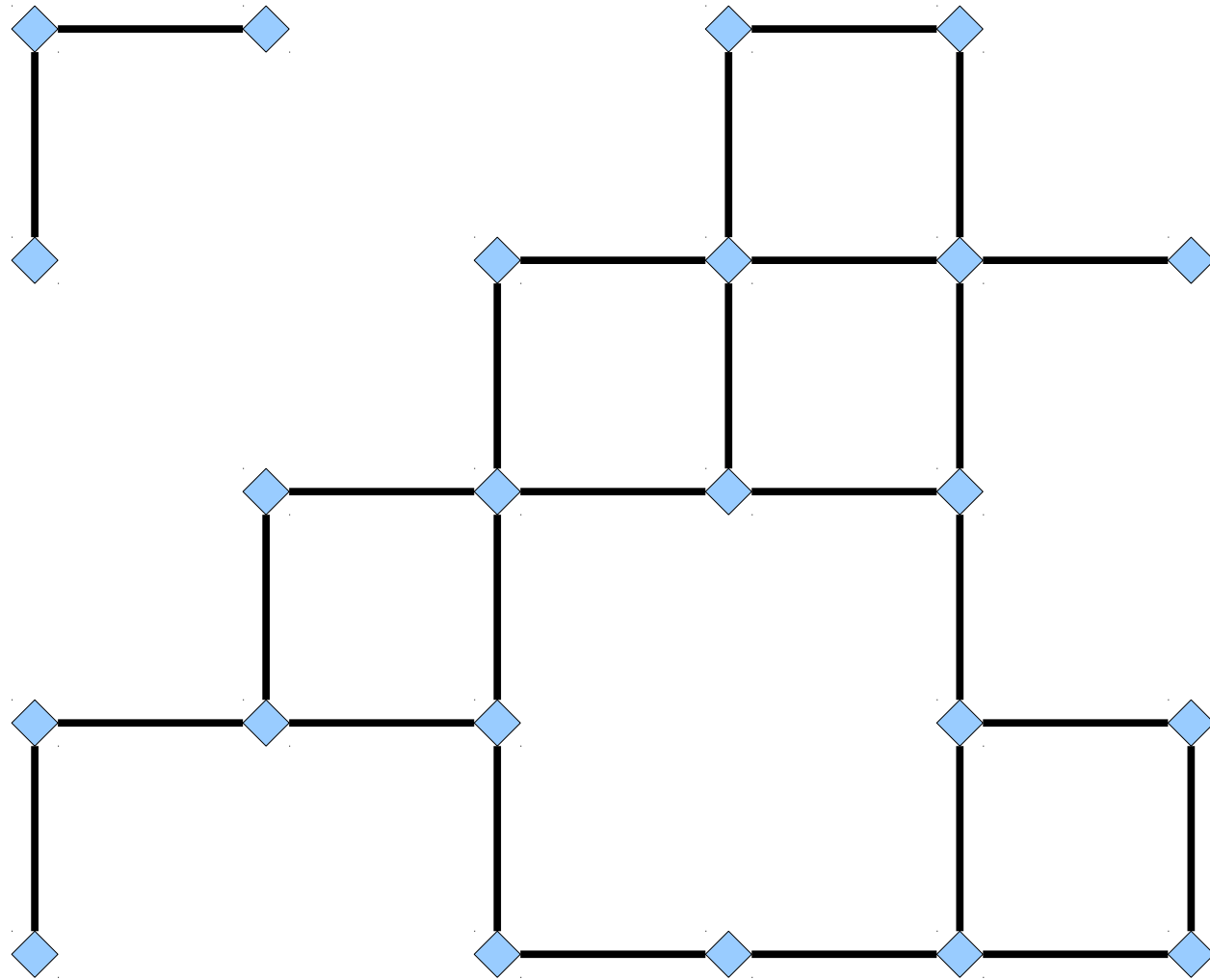
# Solving Domino Tiling



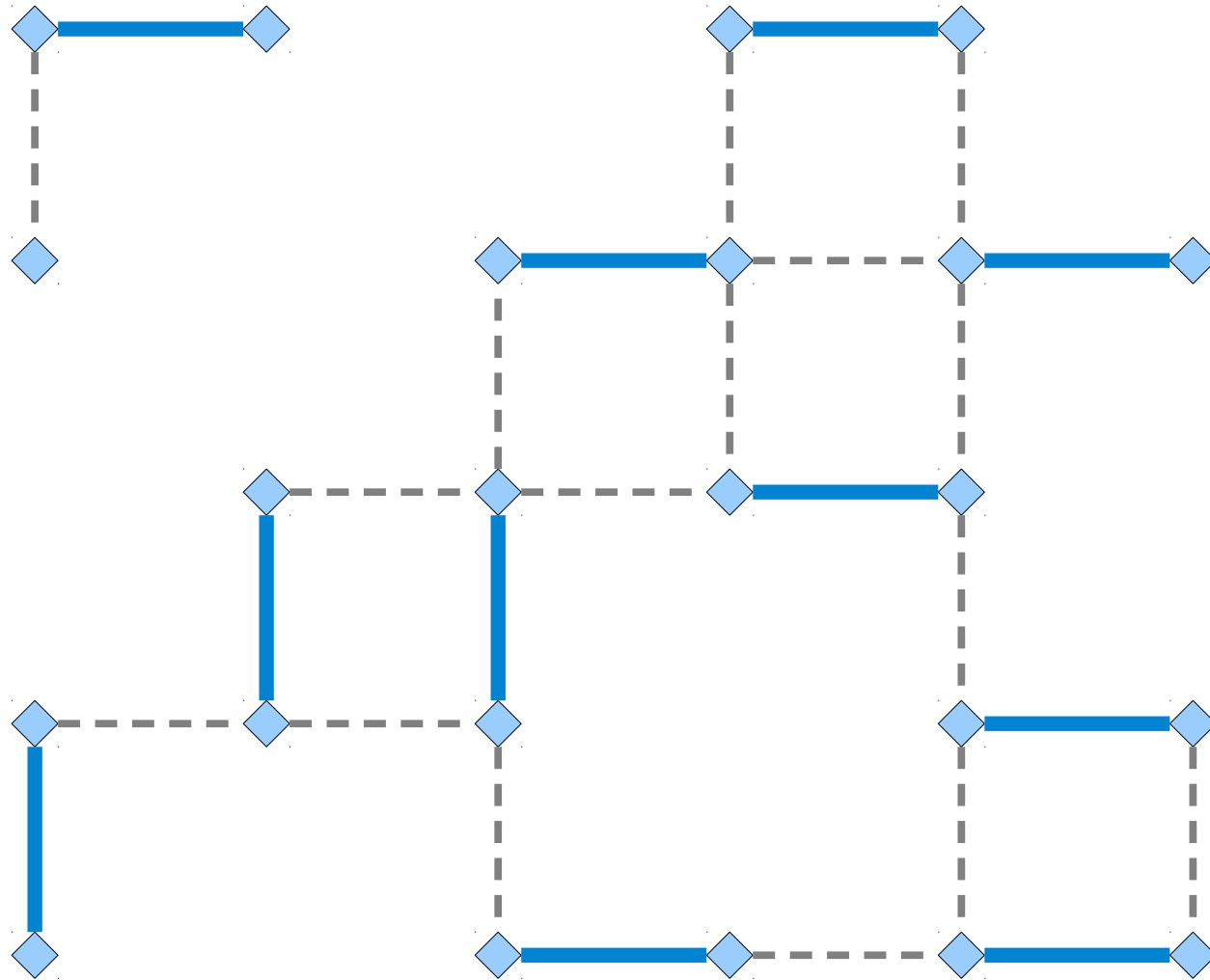
# Solving Domino Tiling



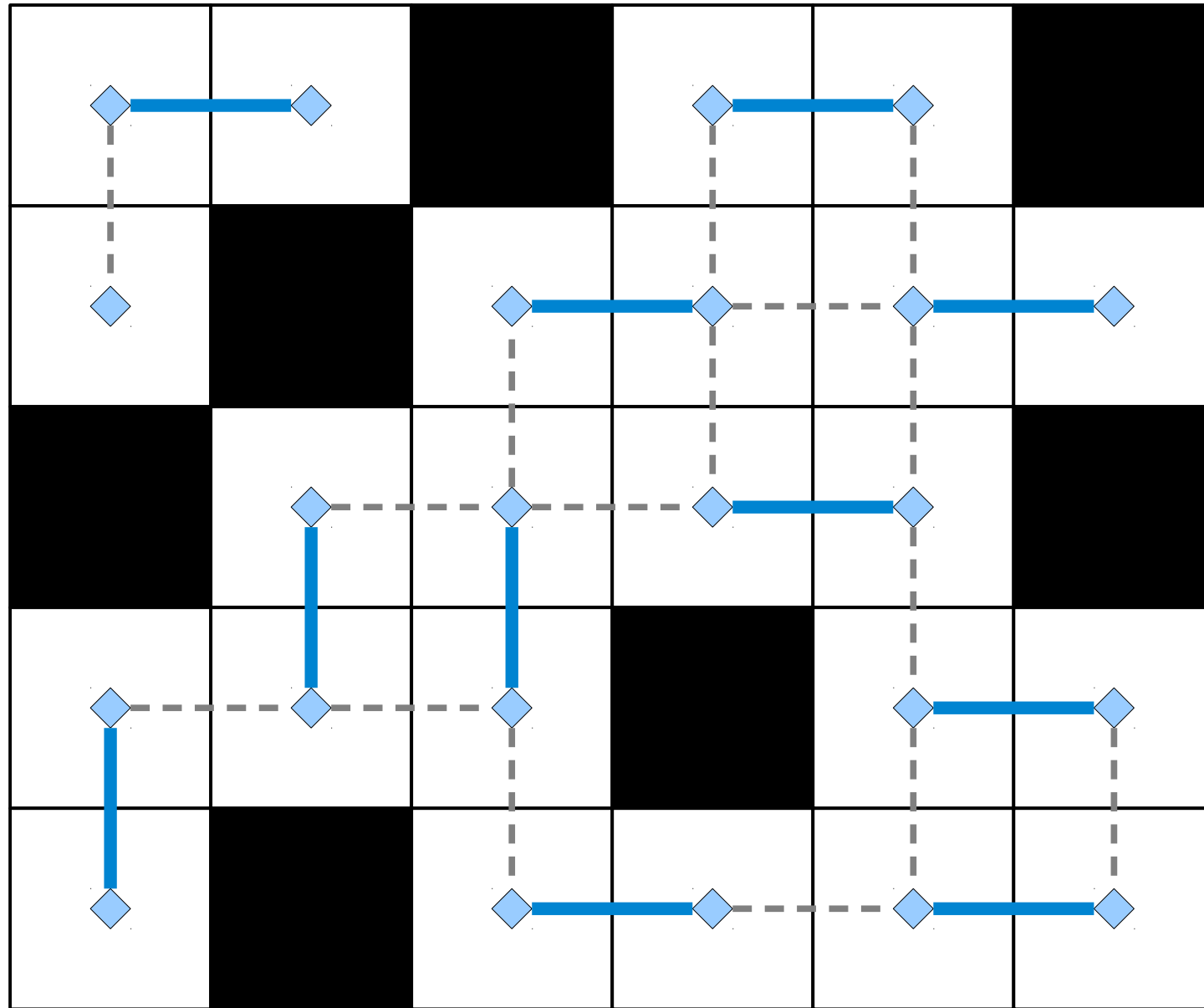
# Solving Domino Tiling



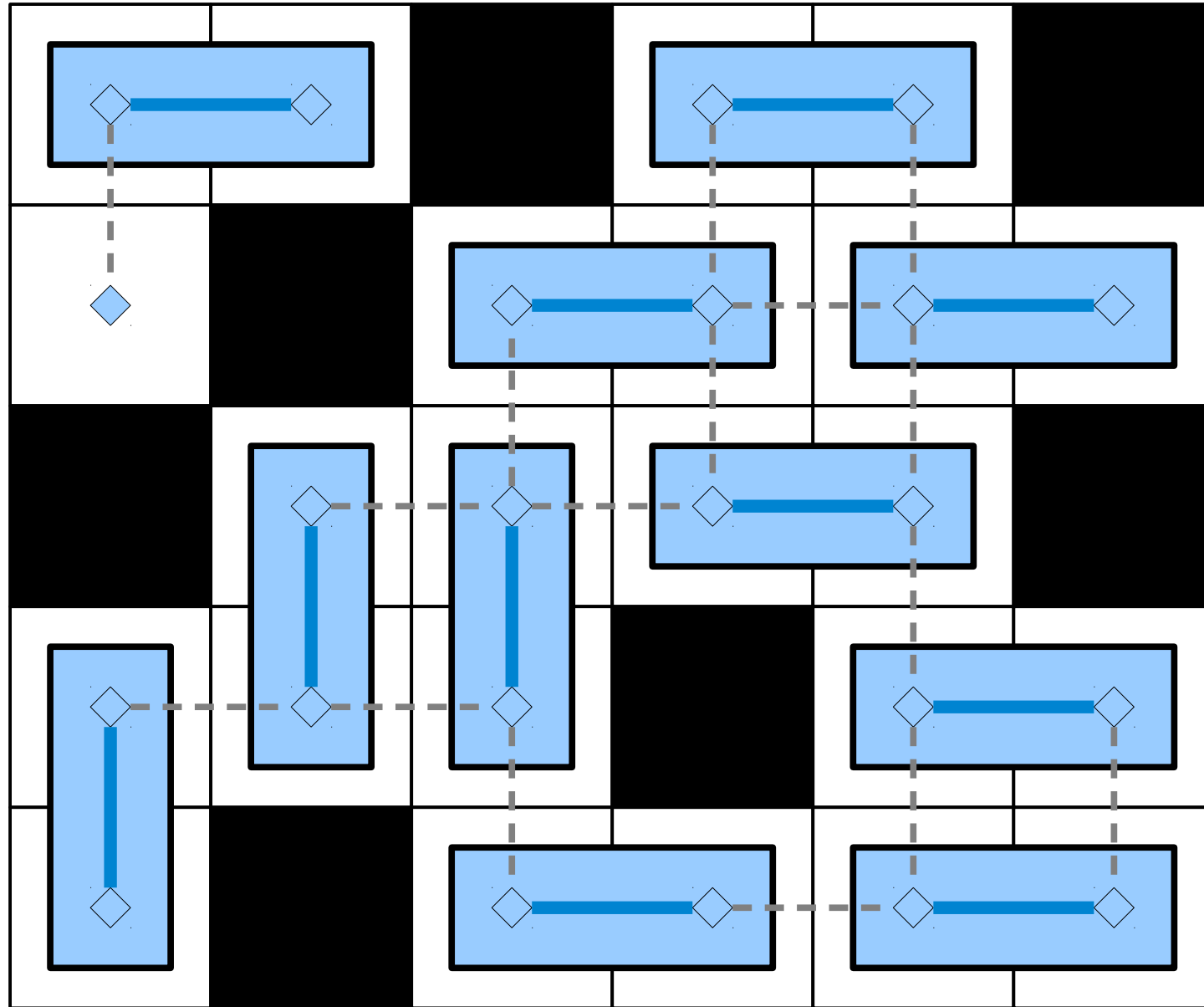
# Solving Domino Tiling



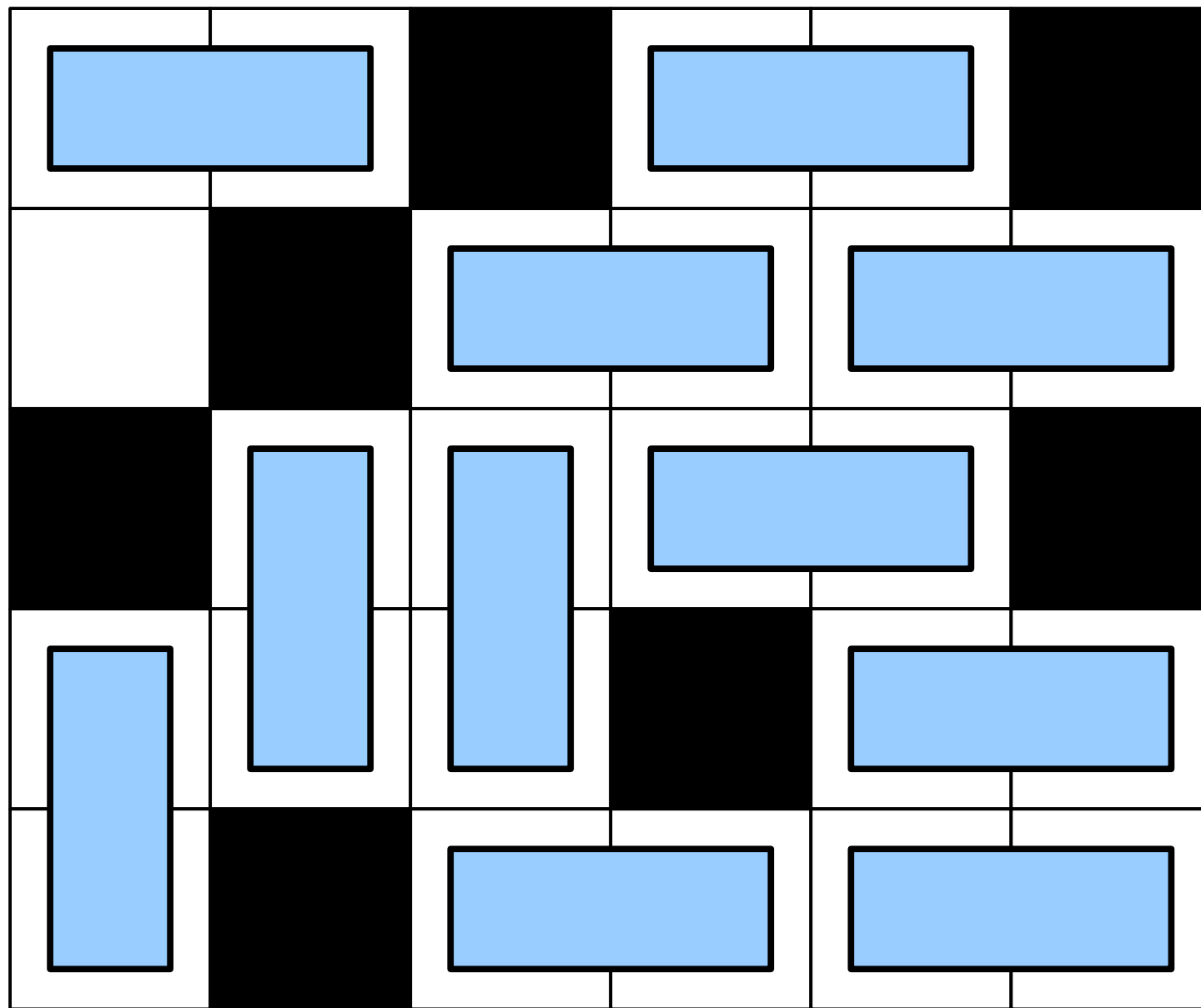
# Solving Domino Tiling



# Solving Domino Tiling



# Solving Domino Tiling



# In Pseudocode

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

Based on this connection between maximum matching and domino tiling, which of the following statements would be more proper to conclude?

- A. Finding a maximum matching isn't any more difficult (in BigO/P-NP terms) than tiling a grid with dominoes.
- B. Tiling a grid with dominoes isn't any more difficult (in BigO/P-NP terms) than finding a maximum matching.

## ***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.

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

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

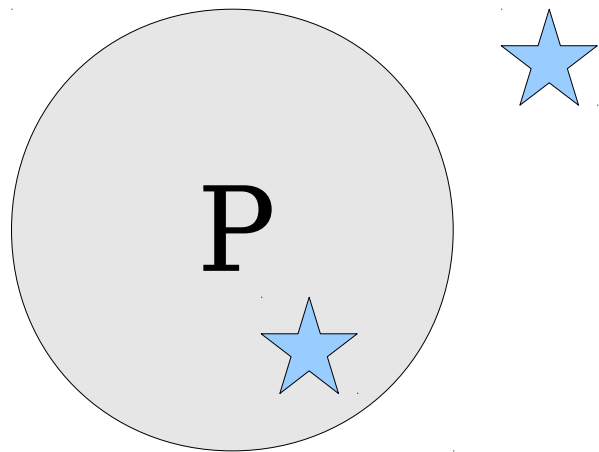
$$A \leq_p B.$$

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

\* Assuming that transform runs in polynomial time.

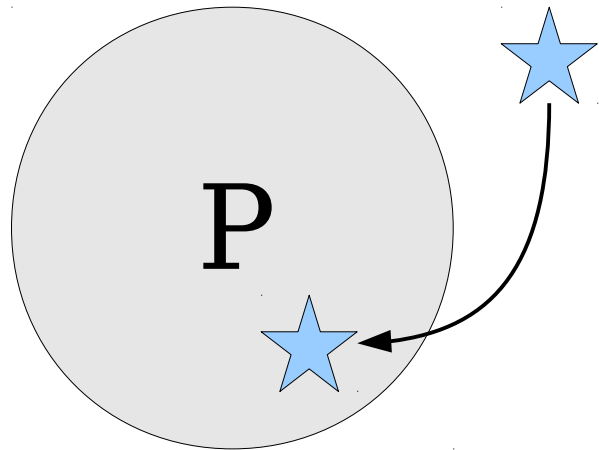
# Polynomial-Time Reductions

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



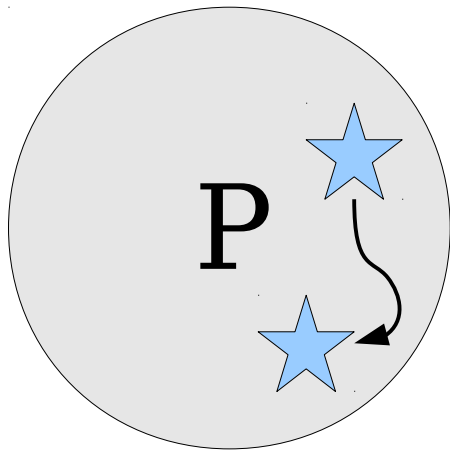
# Polynomial-Time Reductions

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



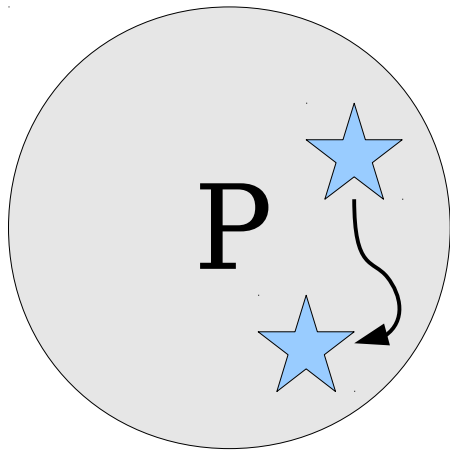
# Polynomial-Time Reductions

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



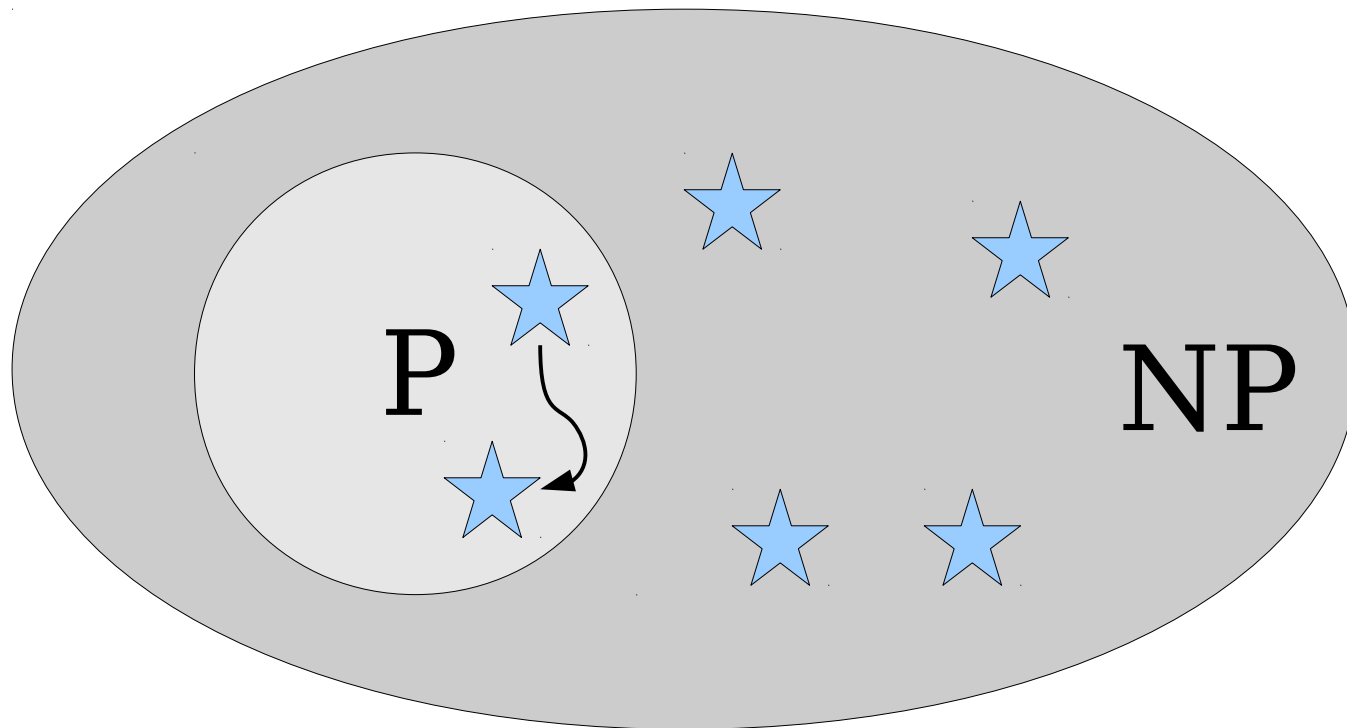
# Polynomial-Time Reductions

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



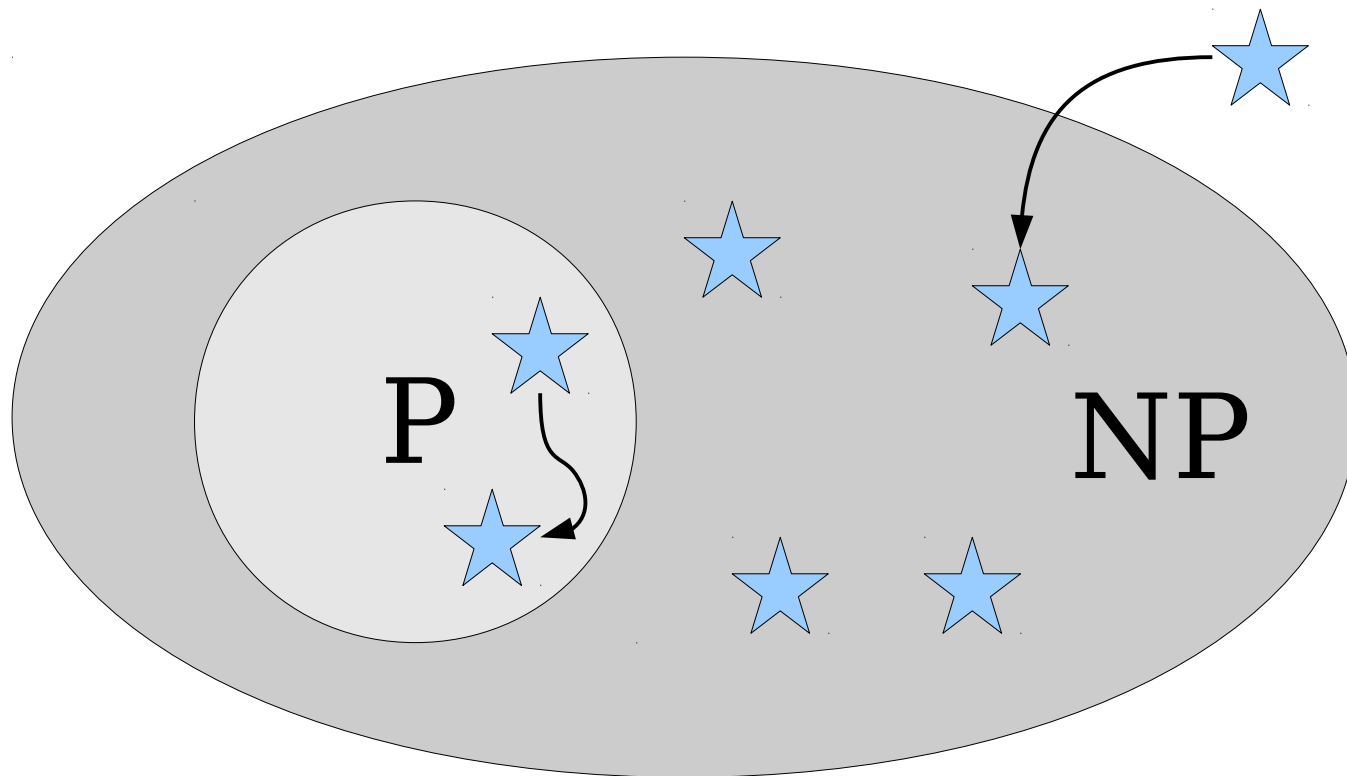
# Polynomial-Time Reductions

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



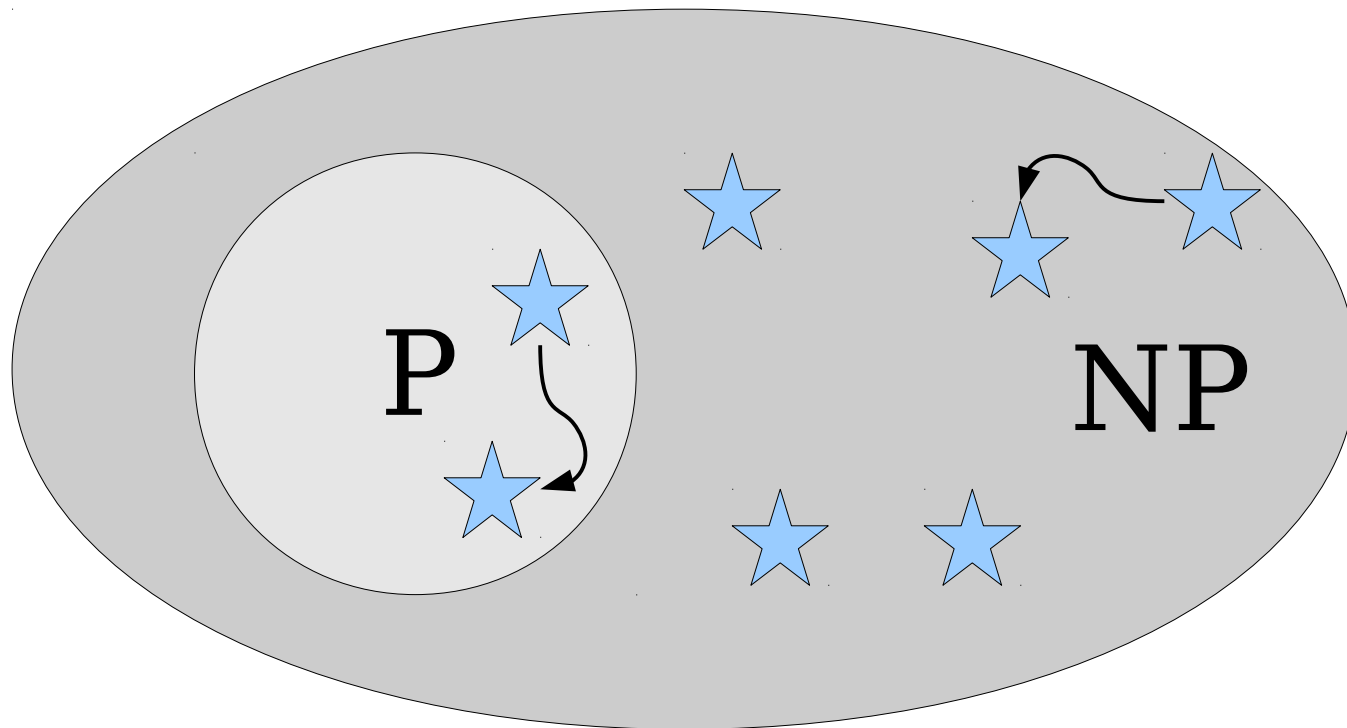
# Polynomial-Time Reductions

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



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

What else can we do with it?

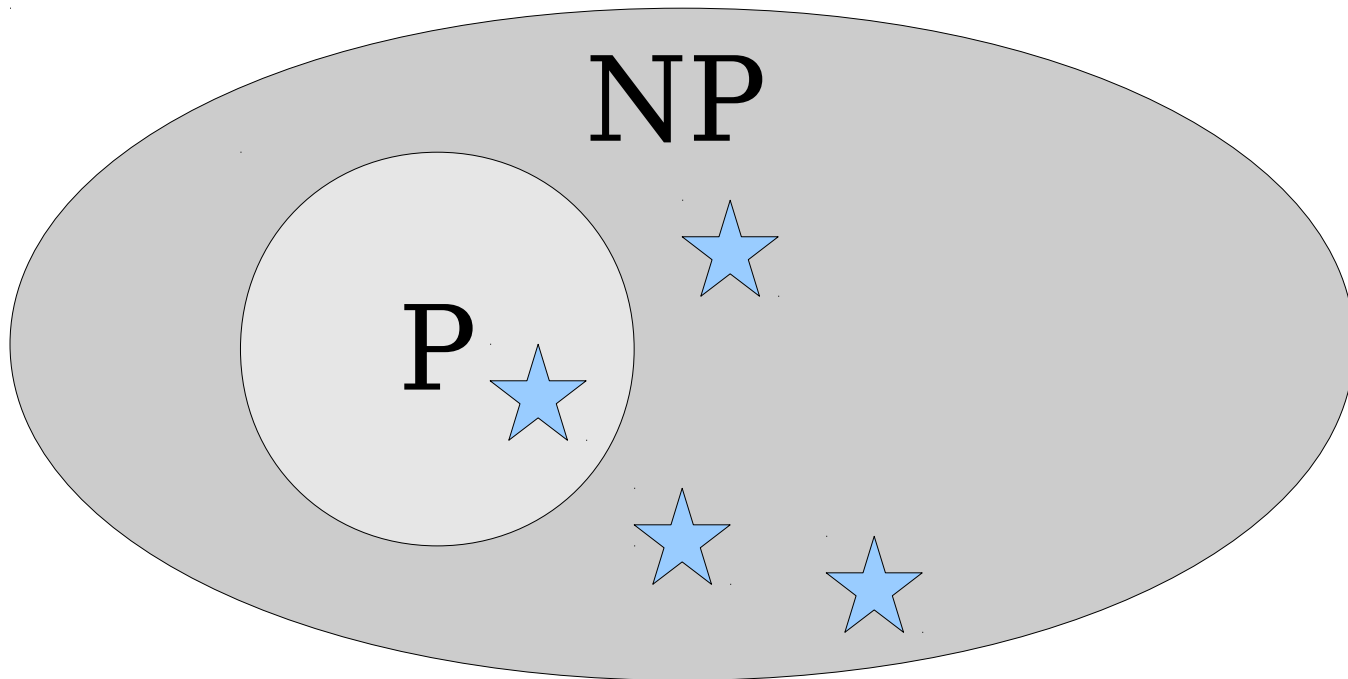
# **NP**-Hardness and **NP**-Completeness

***Question:*** What makes a problem  
hard to solve?

***Intuition:*** To show that some problem is hard, show that lots of other problems reduce to it.

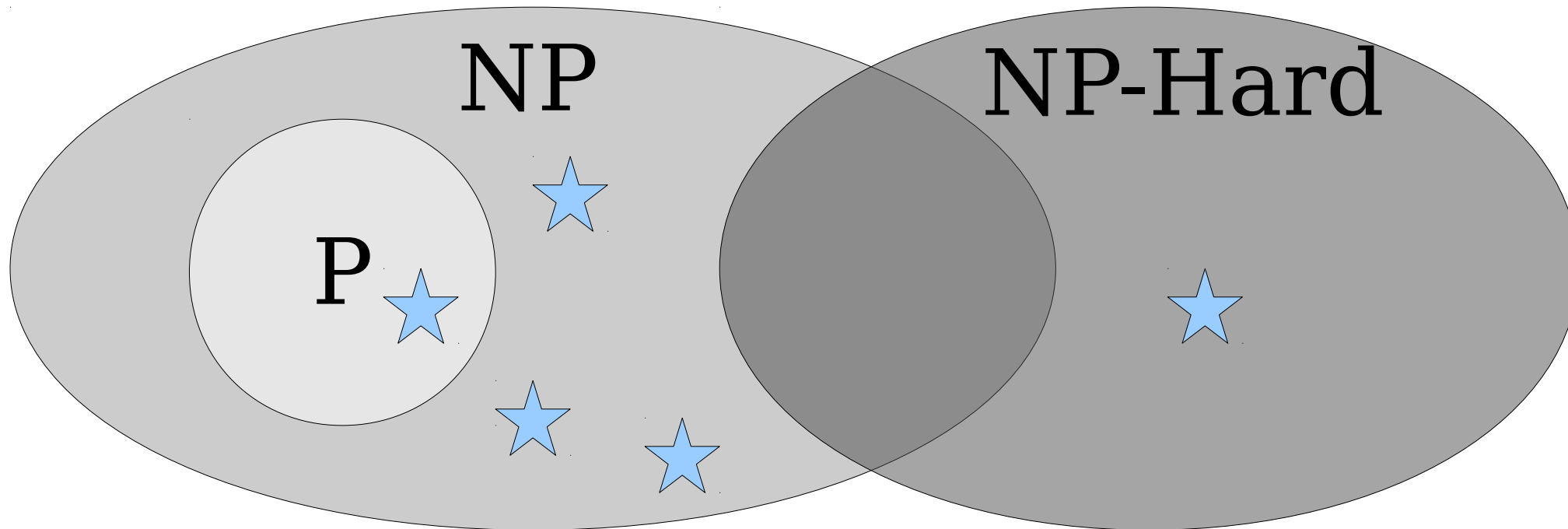
# NP-Hardness

- A language  $L$  is called **NP-hard** if for *every*  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .



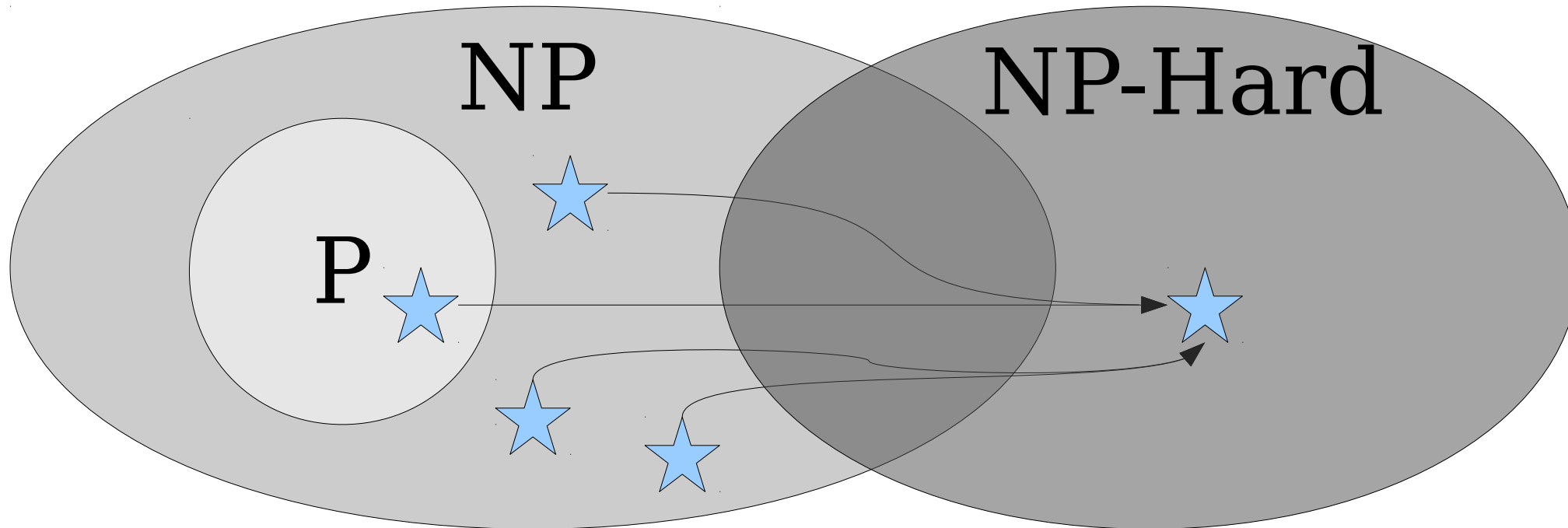
# NP-Hardness

- A language  $L$  is called **NP-hard** if for *every*  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .



# NP-Hardness

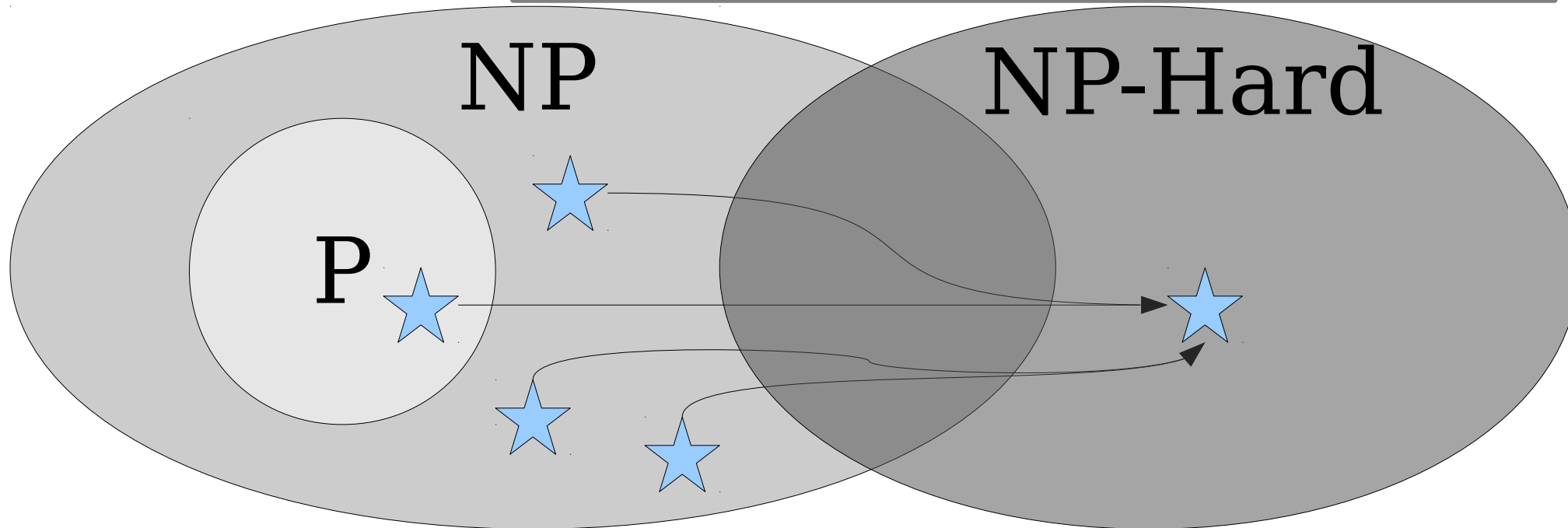
- A language  $L$  is called **NP-hard** if for *every*  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .



# NP-Hardness

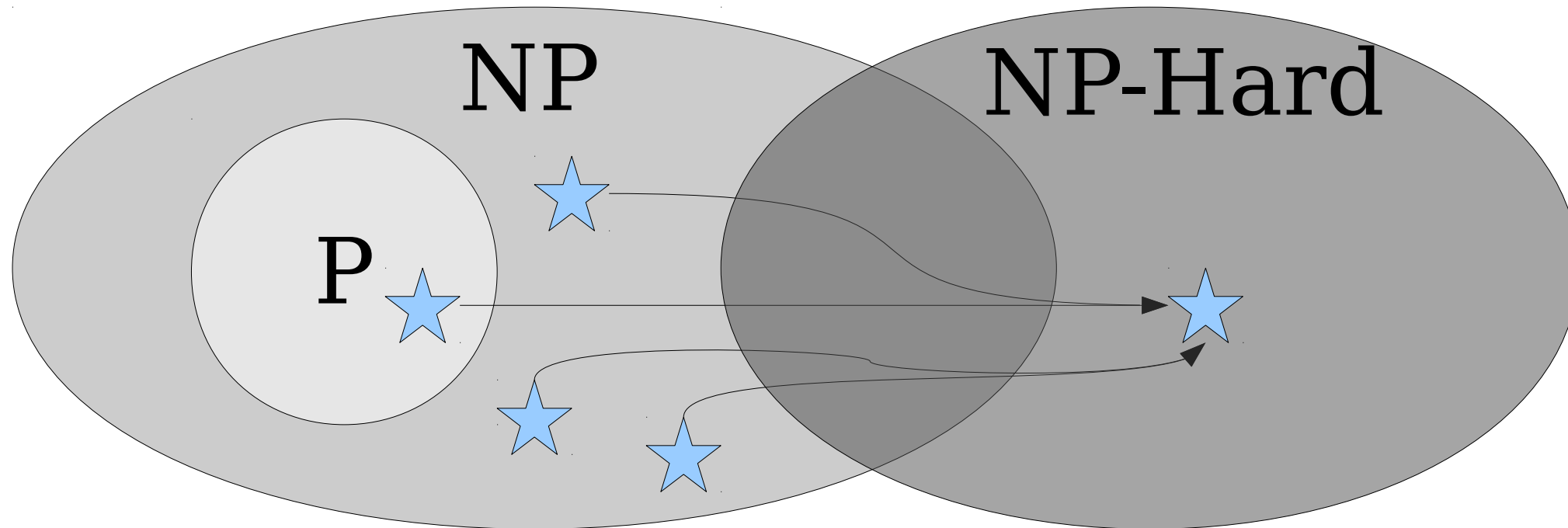
- A language  $L$  is called **NP-hard** if for every  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .

Intuitively:  $L$  has to be at least as hard as every problem in  $\mathbf{NP}$ , since an algorithm for  $L$  can be used to decide all problems in  $\mathbf{NP}$ .



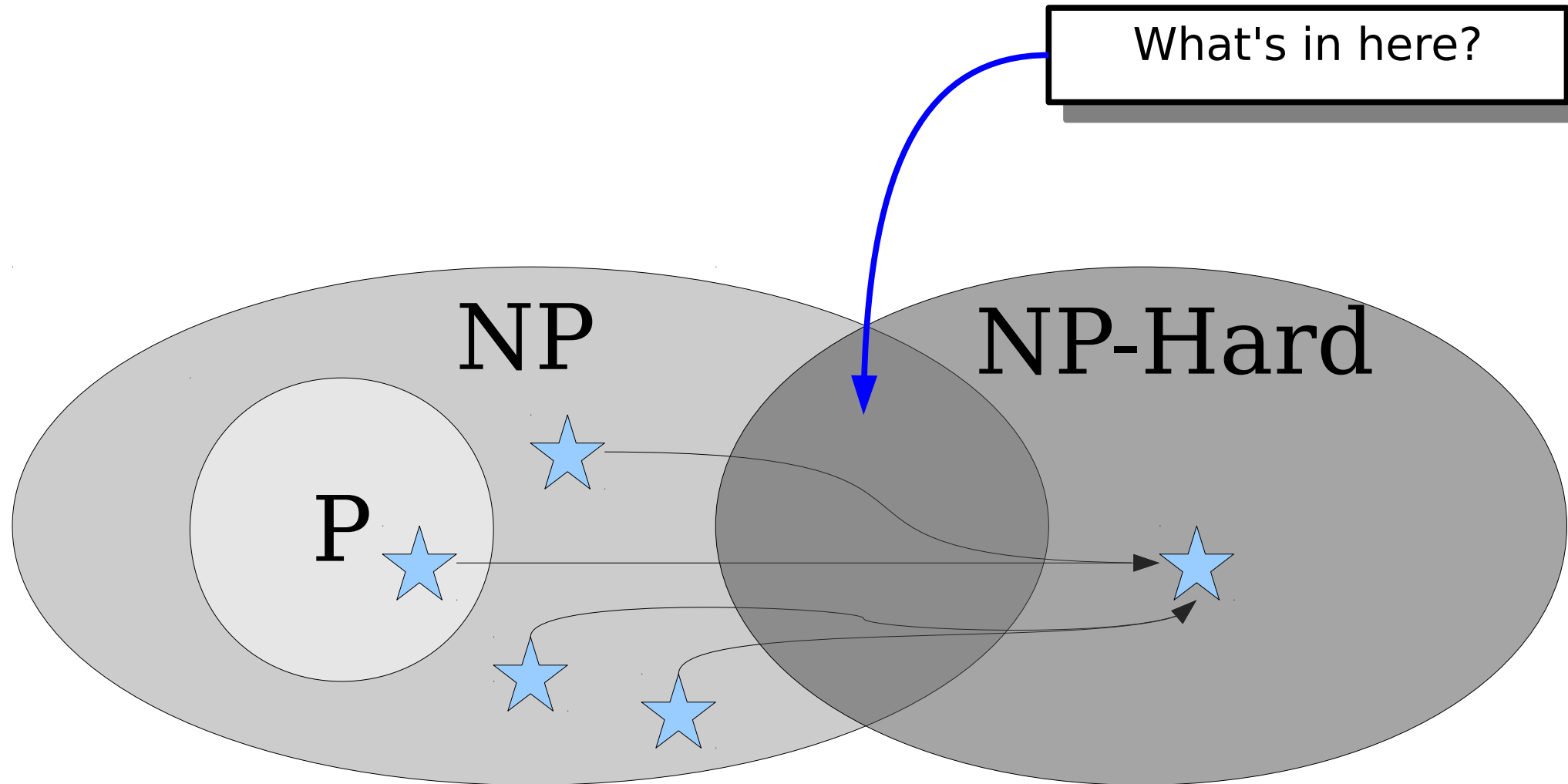
# NP-Hardness

- A language  $L$  is called **NP-hard** if for *every*  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .



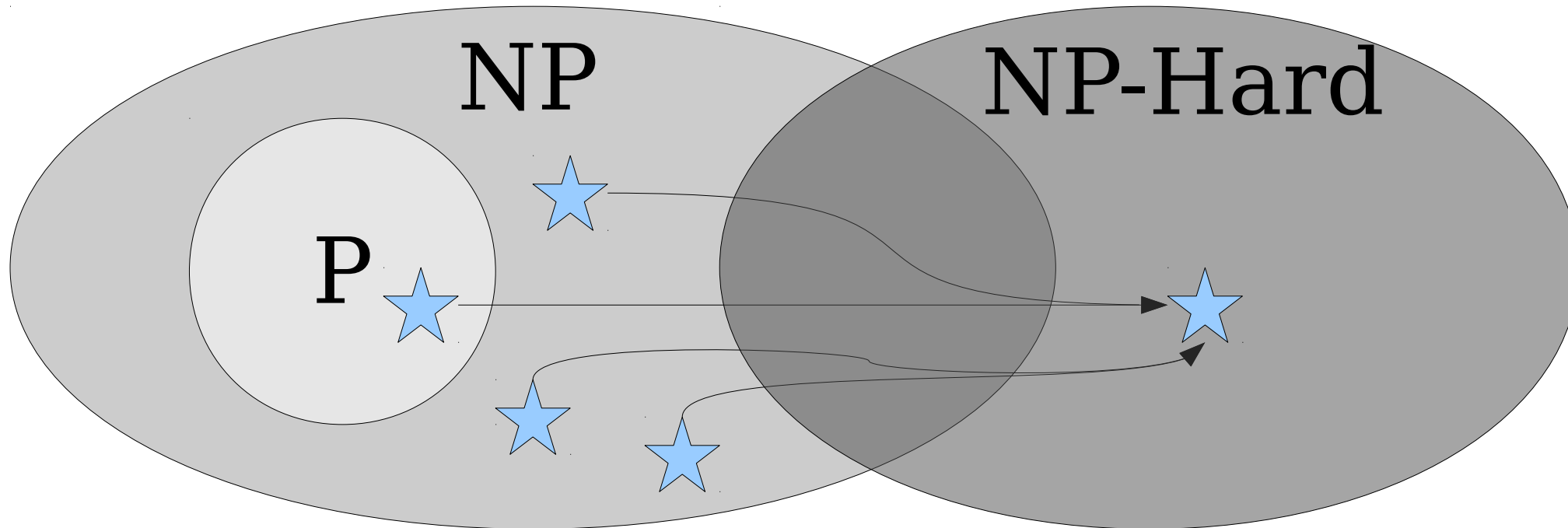
# NP-Hardness

- A language  $L$  is called **NP-hard** if for every  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .



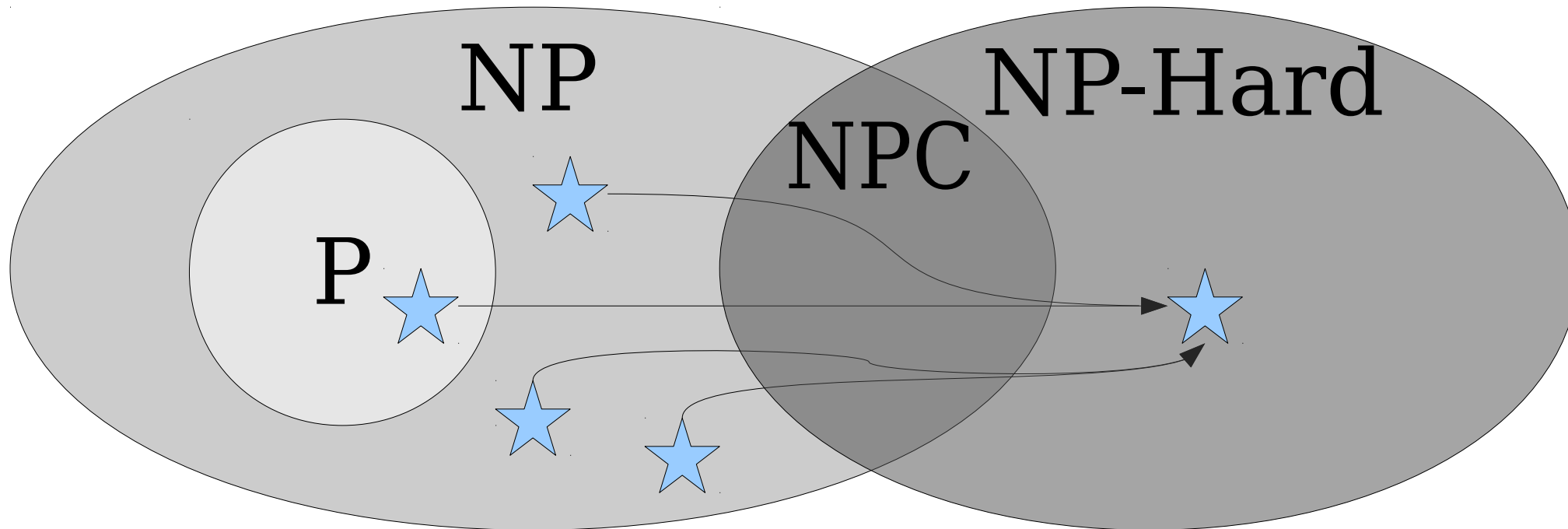
# NP-Hardness

- A language  $L$  is called **NP-hard** if for every  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .
- A language in  $L$  is called **NP-complete** if  $L$  is **NP-hard** and  $L \in \mathbf{NP}$ .
- The class **NPC** is the set of **NP-complete** problems.



# NP-Hardness

- A language  $L$  is called **NP-hard** if for every  $A \in \mathbf{NP}$ , we have  $A \leq_p L$ .
- A language in  $L$  is called **NP-complete** if  $L$  is **NP-hard** and  $L \in \mathbf{NP}$ .
- The class **NPC** is the set of **NP-complete** problems.

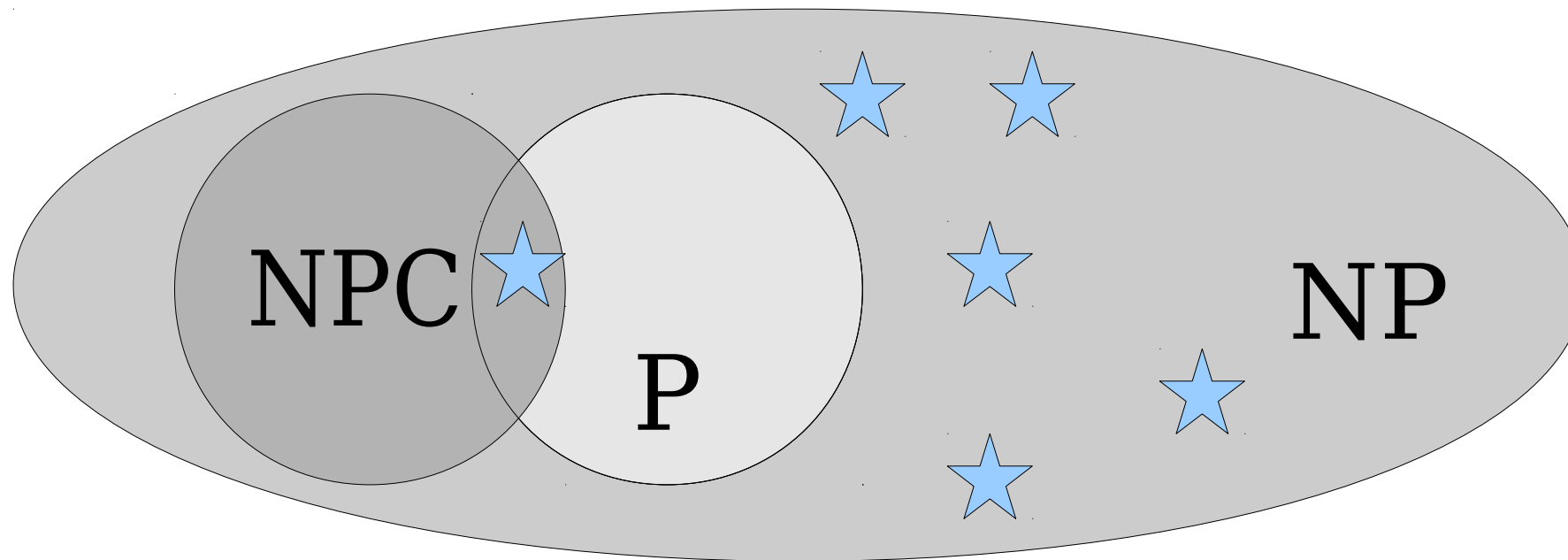


# The Tantalizing Truth

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

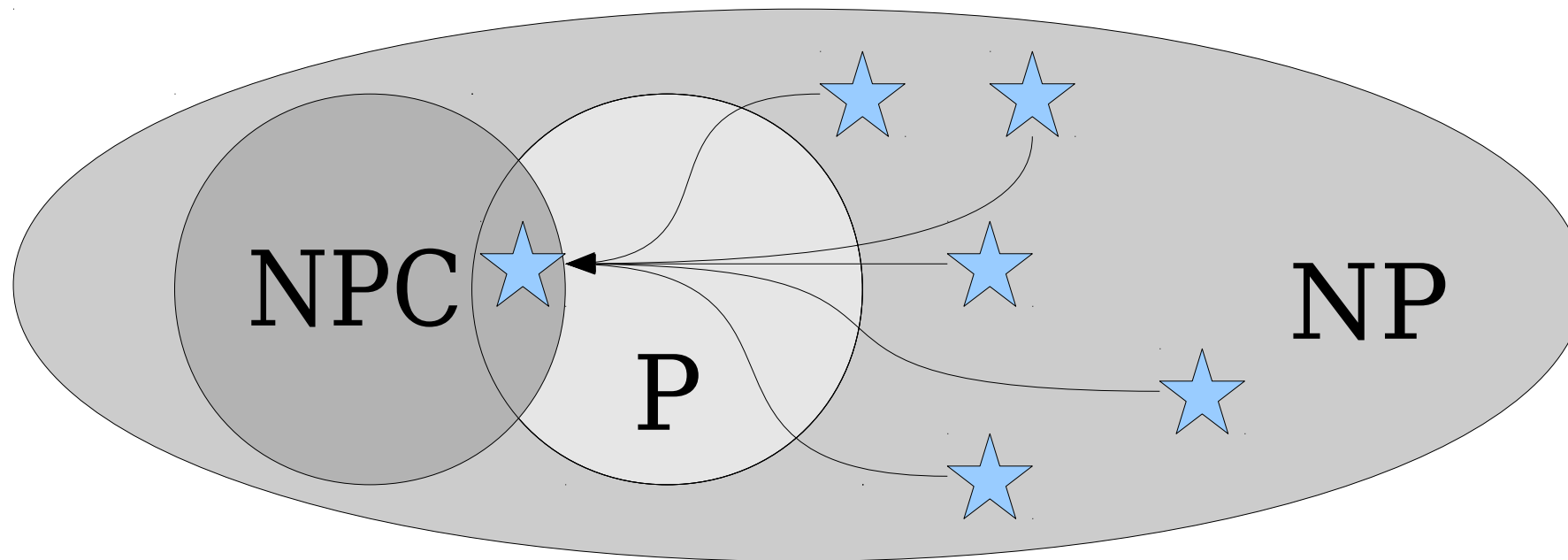
# The Tantalizing Truth

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



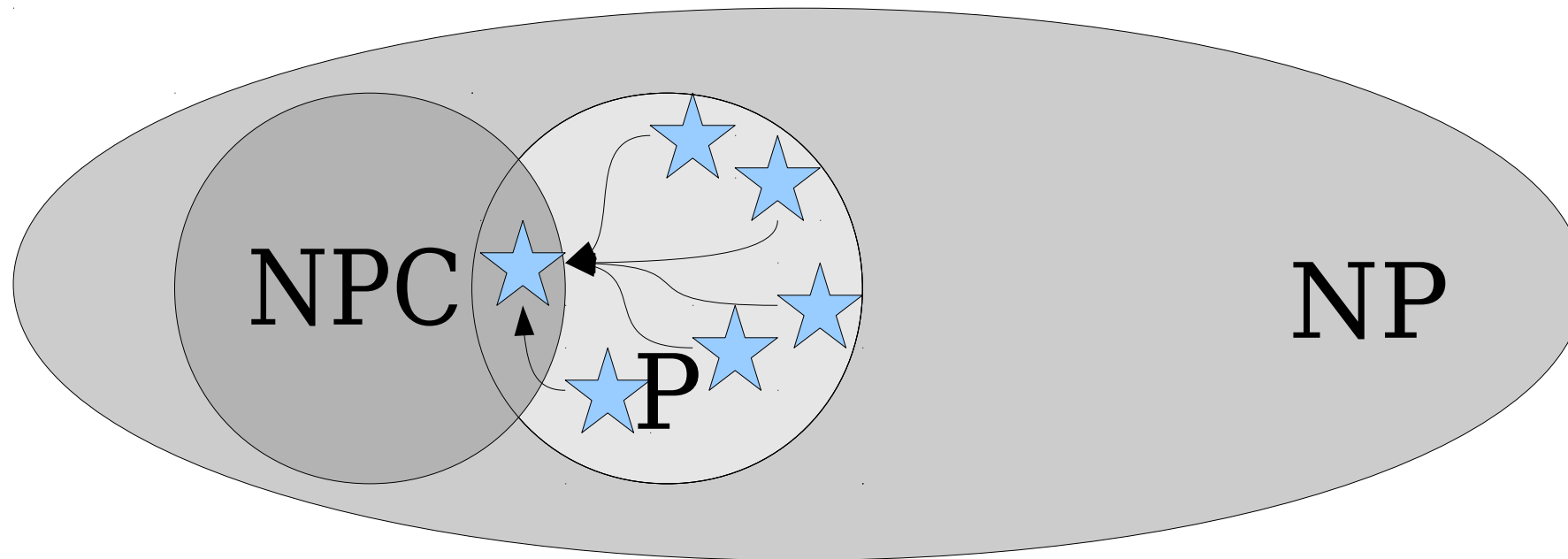
# The Tantalizing Truth

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



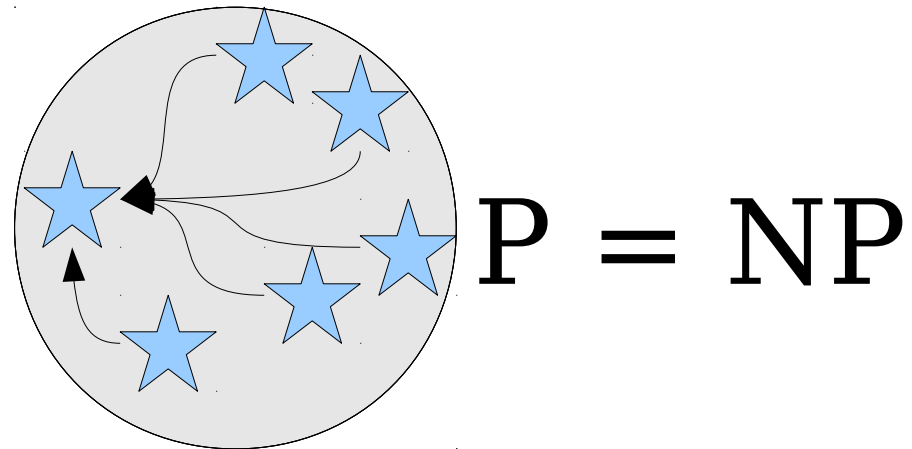
# The Tantalizing Truth

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



# The Tantalizing Truth

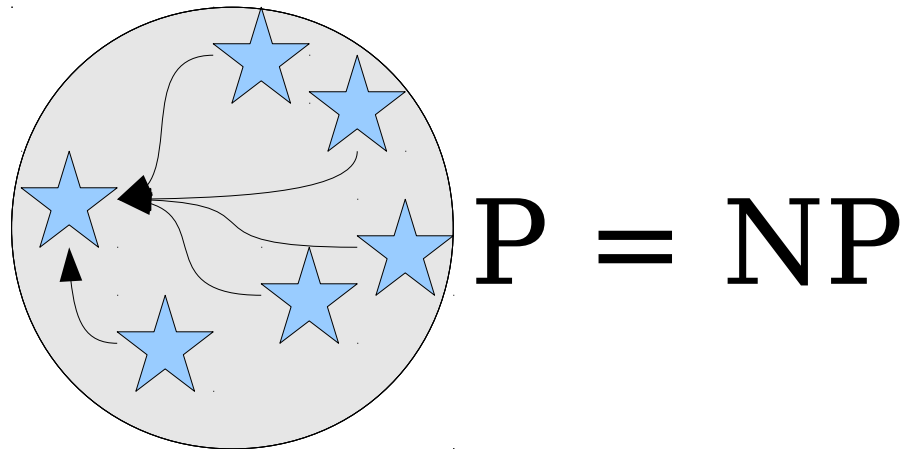
**Theorem:** If *any* NP-complete language is in **P**, then **P** = **NP**.



# The Tantalizing Truth

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

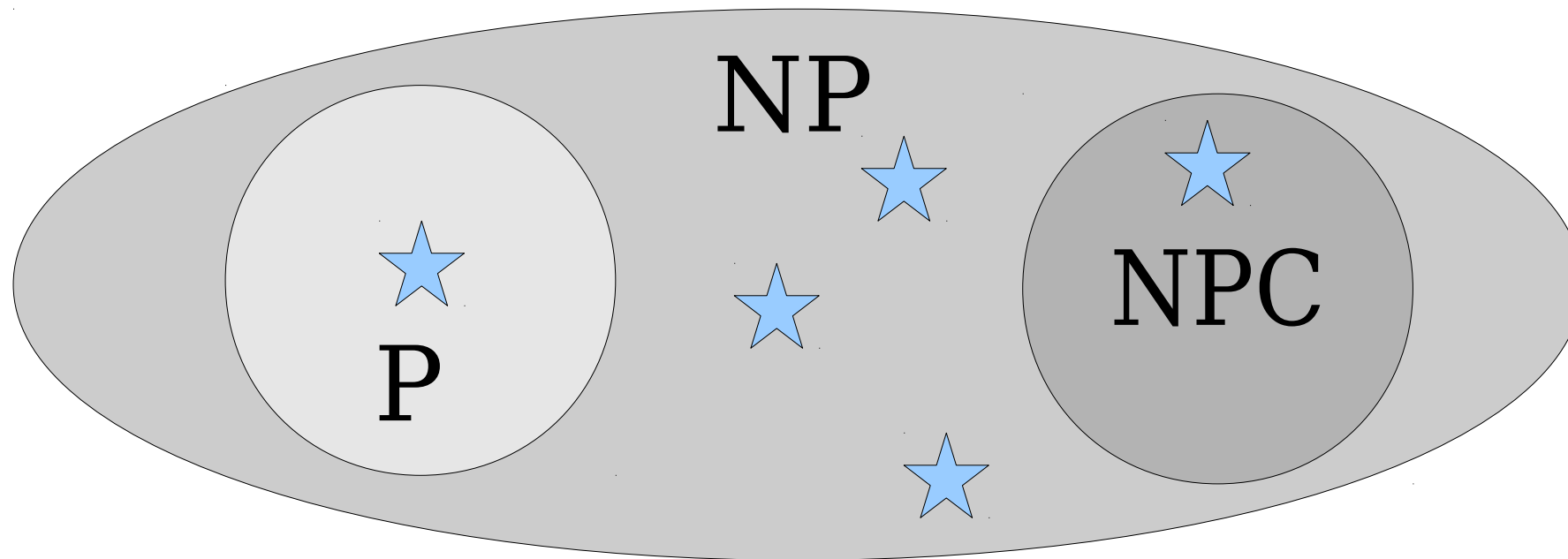
**Proof:** Suppose that  $L$  is **NP**-complete and  $L \in \mathbf{P}$ . Now consider any arbitrary **NP** problem  $A$ . Since  $L$  is **NP**-complete, we know that  $A \leq_p L$ . Since  $L \in \mathbf{P}$  and  $A \leq_p L$ , we see that  $A \in \mathbf{P}$ . Since our choice of  $A$  was arbitrary,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



# The Tantalizing Truth

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

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



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

# Satisfiability

- A propositional logic formula  $\varphi$  is called **satisfiable** if there is some assignment to its variables that makes it evaluate to true.
  - $p \wedge q$  is satisfiable.
  - $p \wedge \neg p$  is unsatisfiable.
  - $p \rightarrow (q \wedge \neg q)$  is satisfiable.
- An assignment of true and false to the variables of  $\varphi$  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  $\varphi$ , is  $\varphi$  satisfiable?**

- Formally:

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

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

***Proof Idea:*** To see that **SAT**  $\in$  **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 polynomial-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.
  - If  $\text{SAT} \in \mathbf{P}$ , then  $\mathbf{P} = \mathbf{NP}$ .
  - If  $\text{SAT} \notin \mathbf{P}$ , then  $\mathbf{P} \neq \mathbf{NP}$ .

# 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, twoplayer 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 end up with kidneys (*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*)