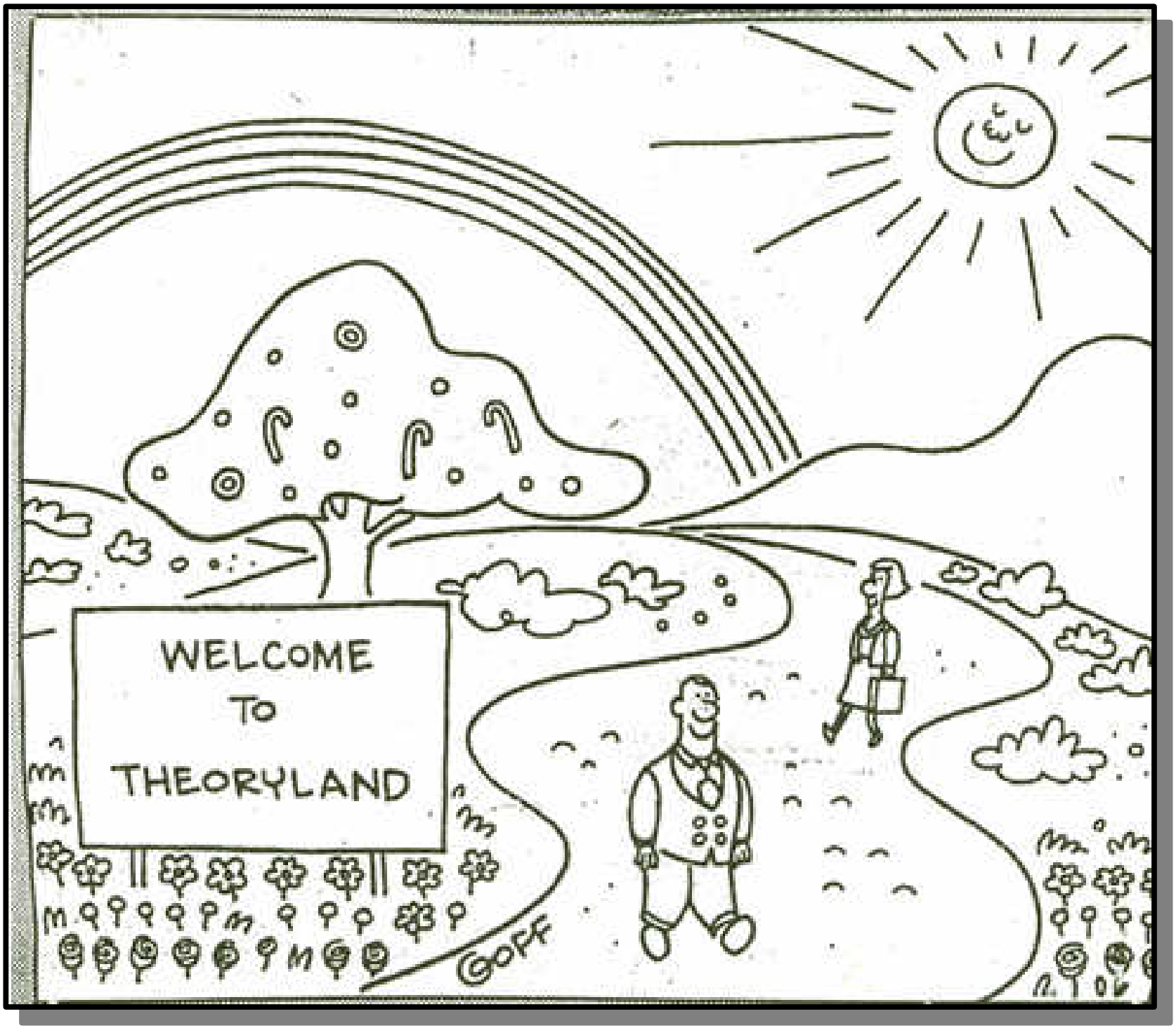
# Complexity Theory

Part One

It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”


WELCOME
TO
THEORYLAND
GOFF

It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

It may be that since one is customarily concerned with existence, [...] decidability, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-decidable* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

# A Decidable Problem

- ***Presburger arithmetic*** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \rightarrow x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \wedge \forall y.\ (P(y) \rightarrow P(y + 1))) \rightarrow \forall x.\ P(x)$
- 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 the tape head at least $2^{2^{cn}}$ times on some inputs of length $n$ (for some fixed constant $c$).

# For Reference

- Assume $c = 1$.

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

# For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$

# For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$

# For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$

# For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$
$$2^{2^4} = 65536$$

# For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$
$$2^{2^4} = 65536$$
$$2^{2^5} = 18446744073709551616$$

# For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$
$$2^{2^4} = 65536$$
$$2^{2^5} = 18446744073709551616$$
$$2^{2^6} = 340282366920938463463374607431768211456$$

# The Limits of Decidability

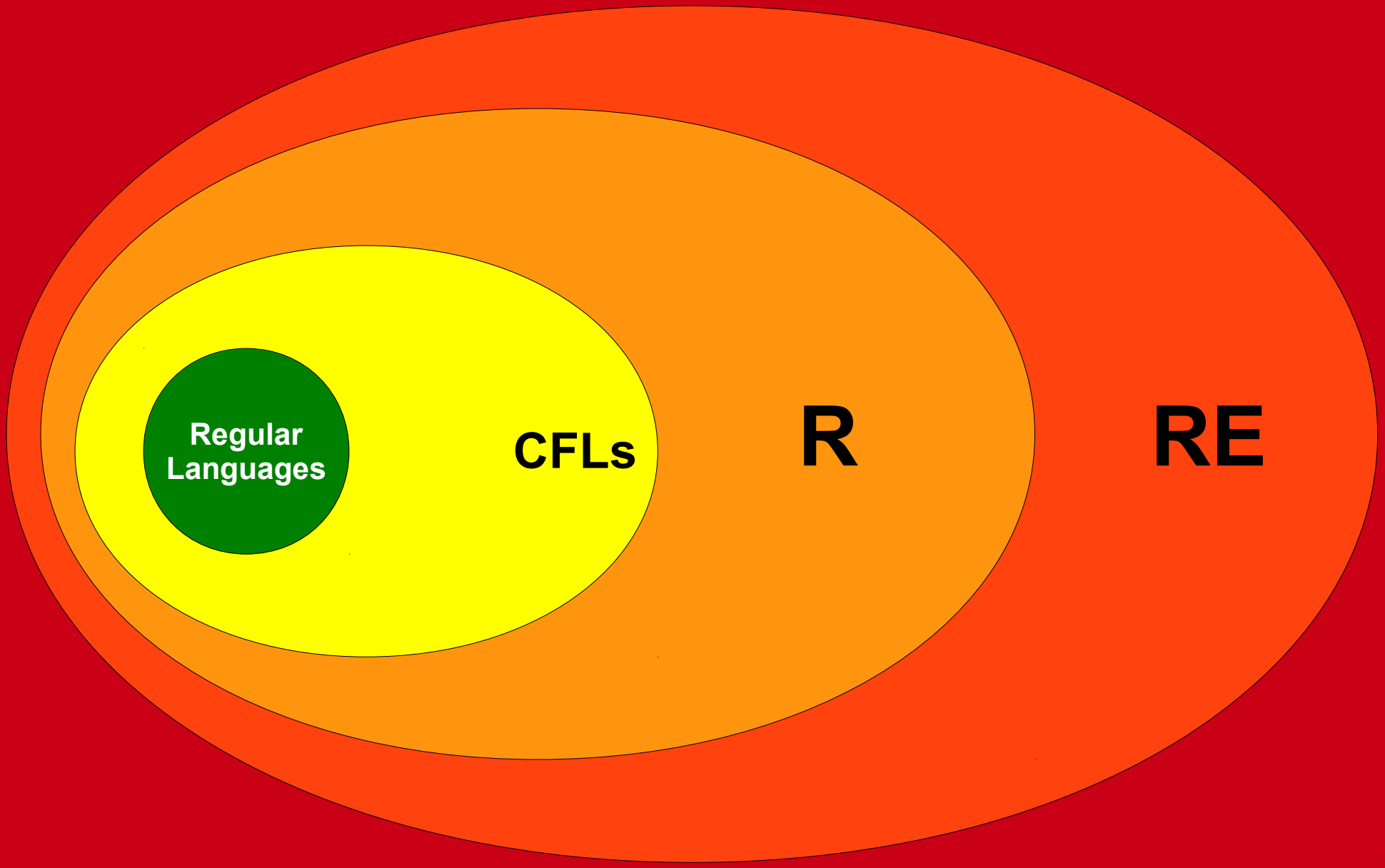
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In ***computability theory***, we ask the question

  What problems can be solved by a computer?

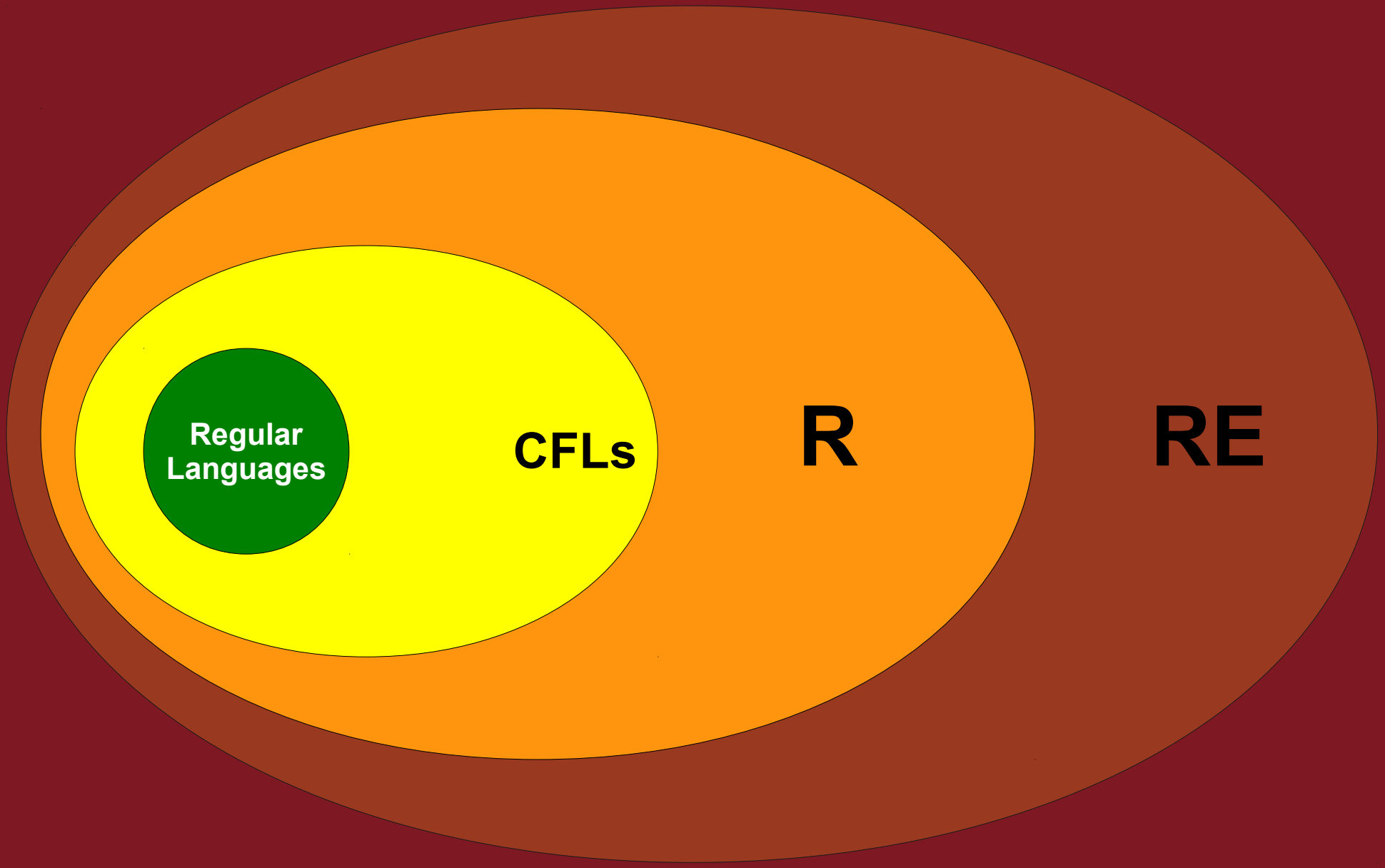
- In ***complexity theory***, we ask the question

  What problems can be solved ***efficiently*** by a computer?

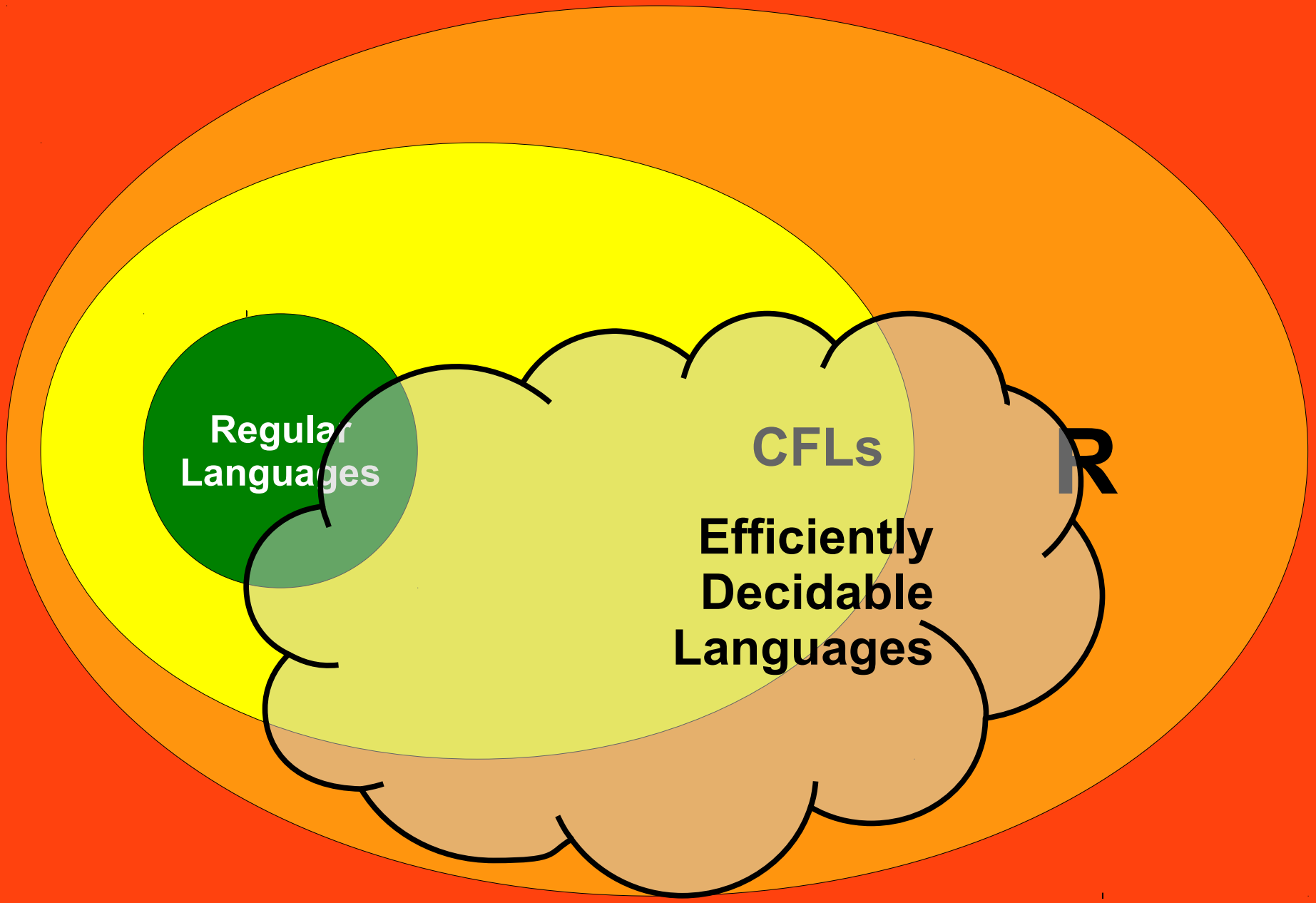
- In the remainder of this course, we will explore this question in more detail.




All Languages




All Languages




Undecidable Languages

# The Setup

- In order to study computability, we needed to answer these questions:
  - What is “computation?”
  - What is a “problem?”
  - What does it mean to “solve” a problem?
- To study complexity, we need to answer these questions:
  - What does “complexity” even mean?
  - What is an “efficient” solution to a problem?

# 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.
- From a decidability perspective, this is totally fine.
- From a complexity perspective, this is totally unacceptable.

# A Sample Problem

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

# A Sample Problem

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

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

# A Sample Problem

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

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

# A Sample Problem

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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.

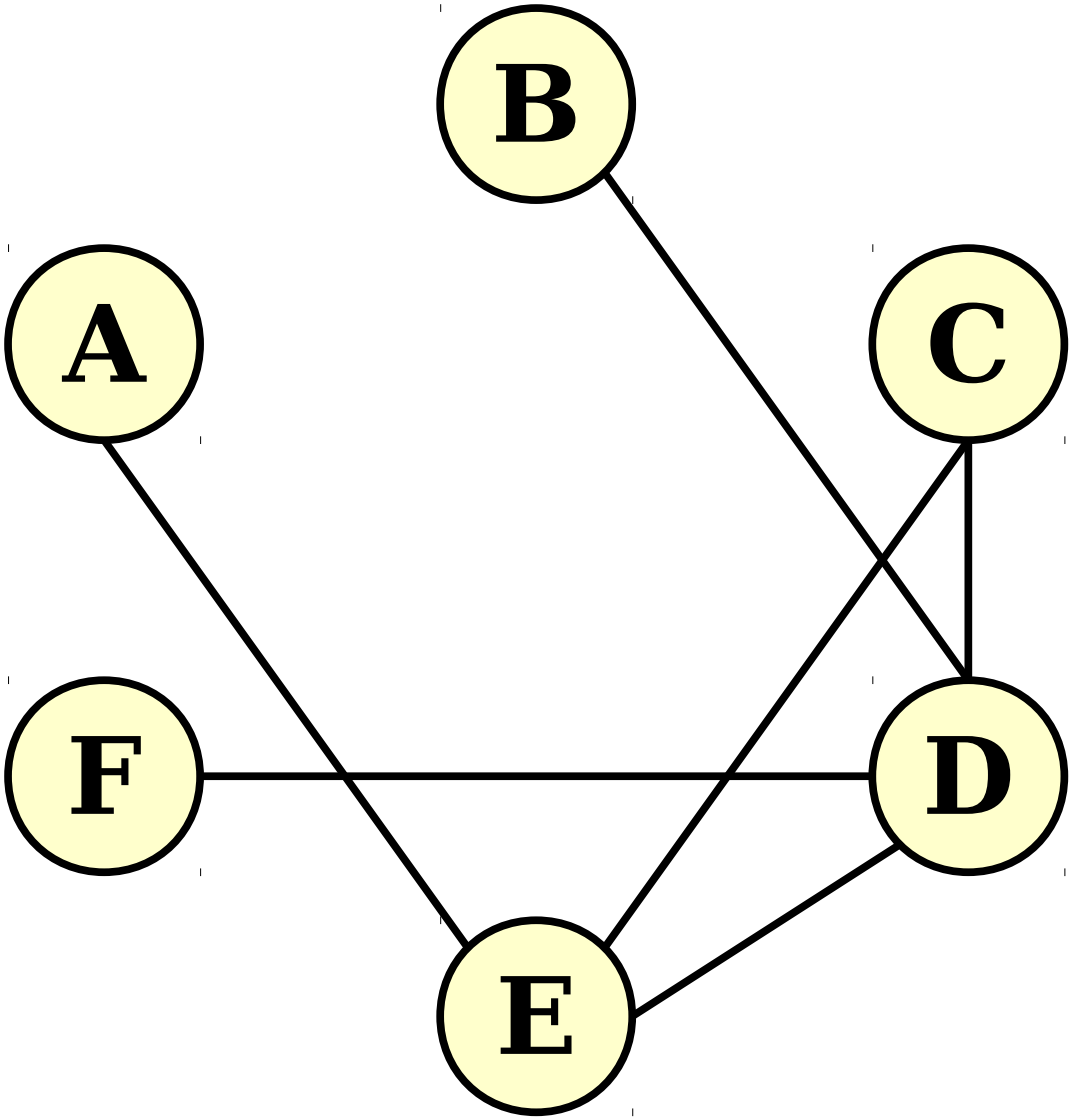
# Longest Increasing Subsequences

- **One possible algorithm:** try all subsequences, find the longest one that's increasing, and return that.
- There are $2^n$ subsequences of an array of length $n$.
  - (Each subset of the elements gives back a subsequence.)
- Checking all of them to find the longest increasing subsequence will take time $O(n \cdot 2^n)$.
- Nifty fact: the age of the universe is about $4.3 \times 10^{26}$ nanoseconds old. That's about $2^{85}$ nanoseconds.
- Practically speaking, this algorithm doesn't terminate if you give it an input of size 100 or more.

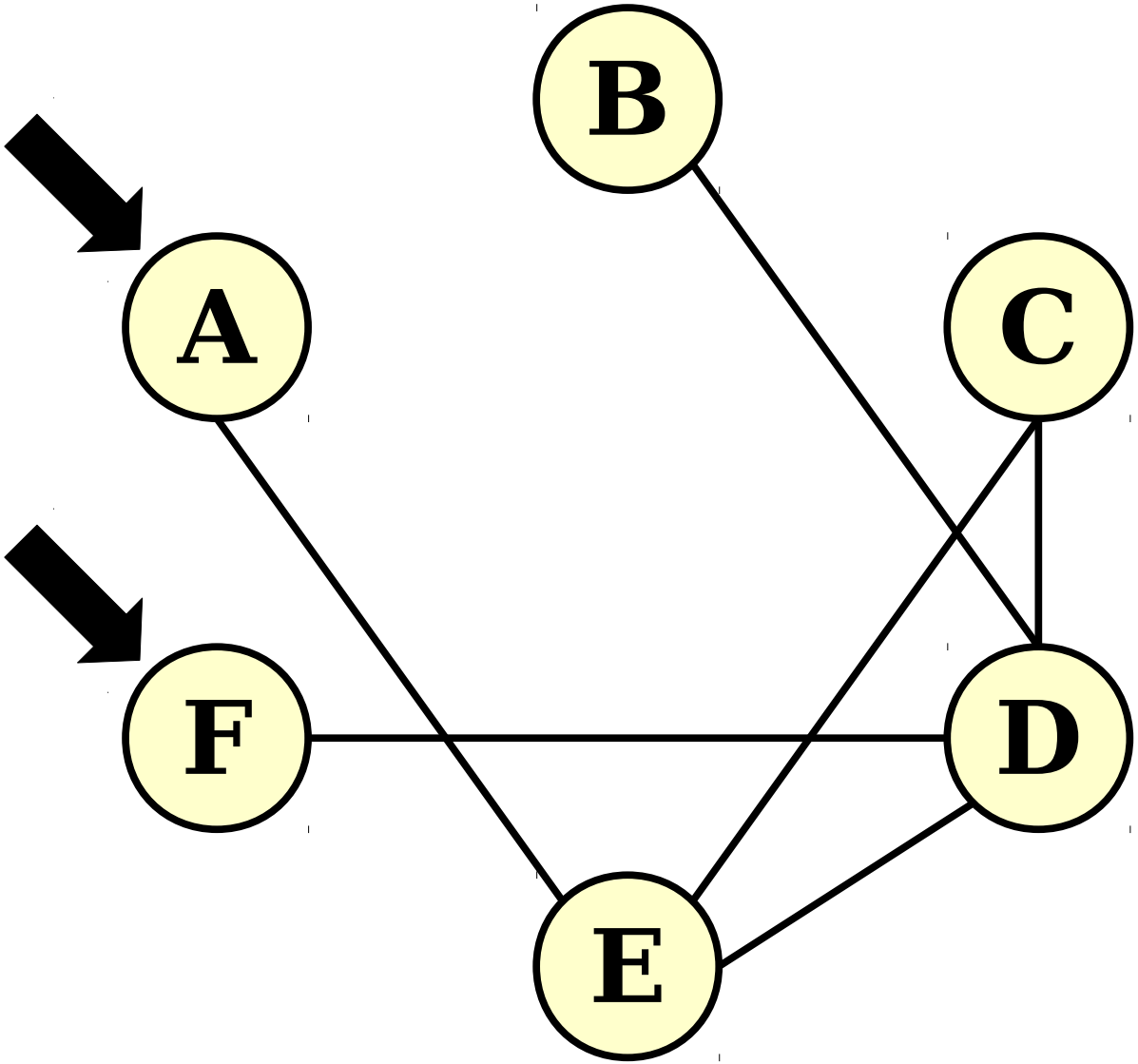
# Longest Increasing Subsequences

- ***Theorem:*** There is an algorithm that can find the longest increasing subsequence of an array in time $O(n \log n)$.
- The algorithm is *beautiful* and surprisingly elegant. Look up ***patience sorting*** if you're curious.
- This algorithm works by exploiting particular aspects of how longest increasing subsequences are constructed. It's not immediately obvious that it works correctly.

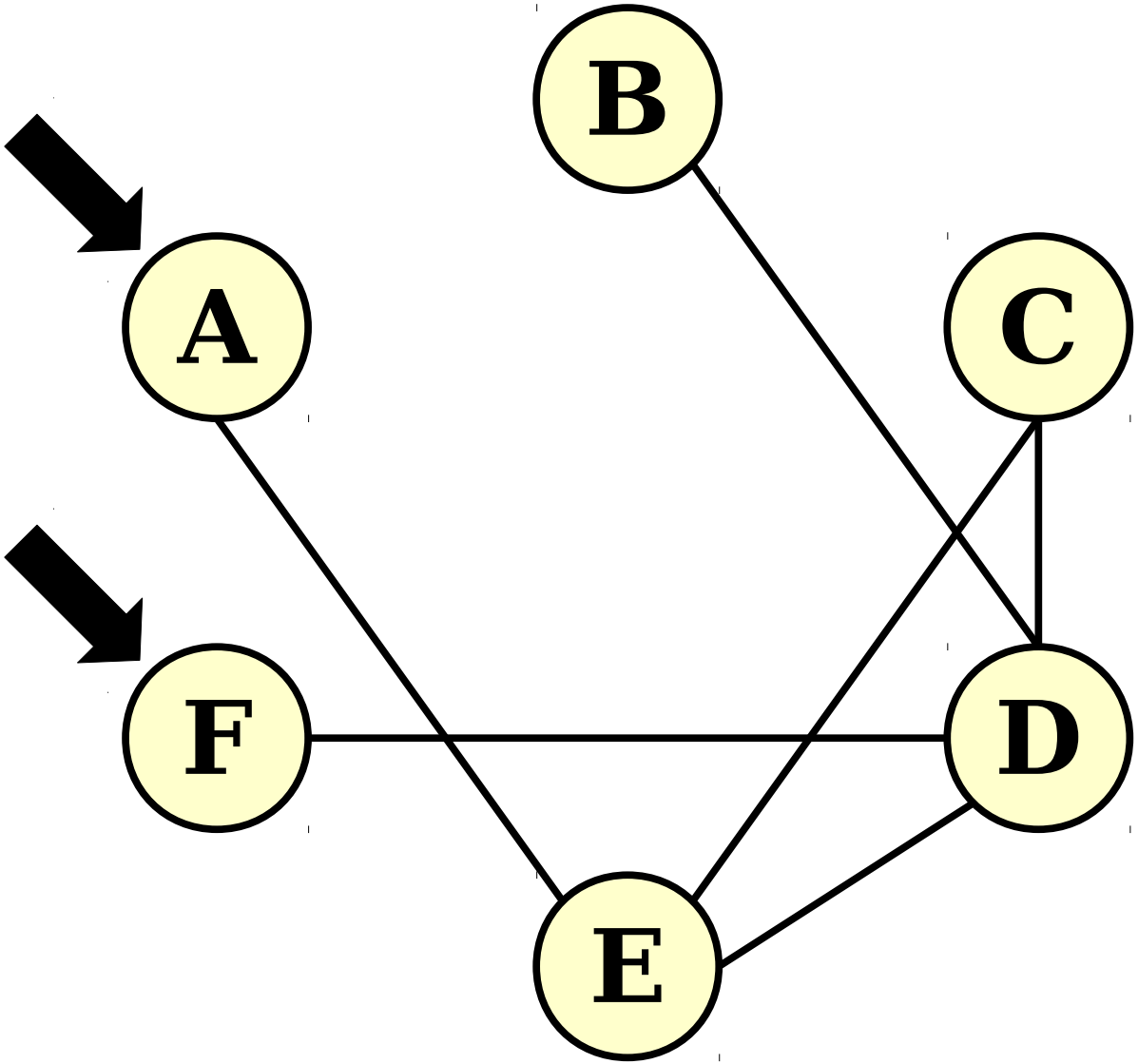
# Another Problem



# Another Problem



# Another Problem




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

# Shortest Paths

- It is possible to find the shortest path in a graph by listing off all sequences of nodes in the graph in ascending order of length and finding the first that's a path.
- This takes time $O(n \cdot n!)$ in an $n$-node graph.
- For reference: 29! nanoseconds is longer than the lifetime of the universe.

# Shortest Paths

- ***Theorem:*** It's possible to find the shortest path between two nodes in an $n$-node, $m$-edge graph in time $O(m + n)$.
- This is the breadth-first search algorithm. Take CS106B/X or CS161 for more details!
- The algorithm is a bit nuanced. It uses some specific properties of shortest paths and the proof of correctness is actually nontrivial.

# For Comparison

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

# Defining Efficiency

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

# Polynomials and Exponentials

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

- 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:
  - $n^{0.0001 \log n}$
  - $1.000000001^n$

# 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 after the other 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**

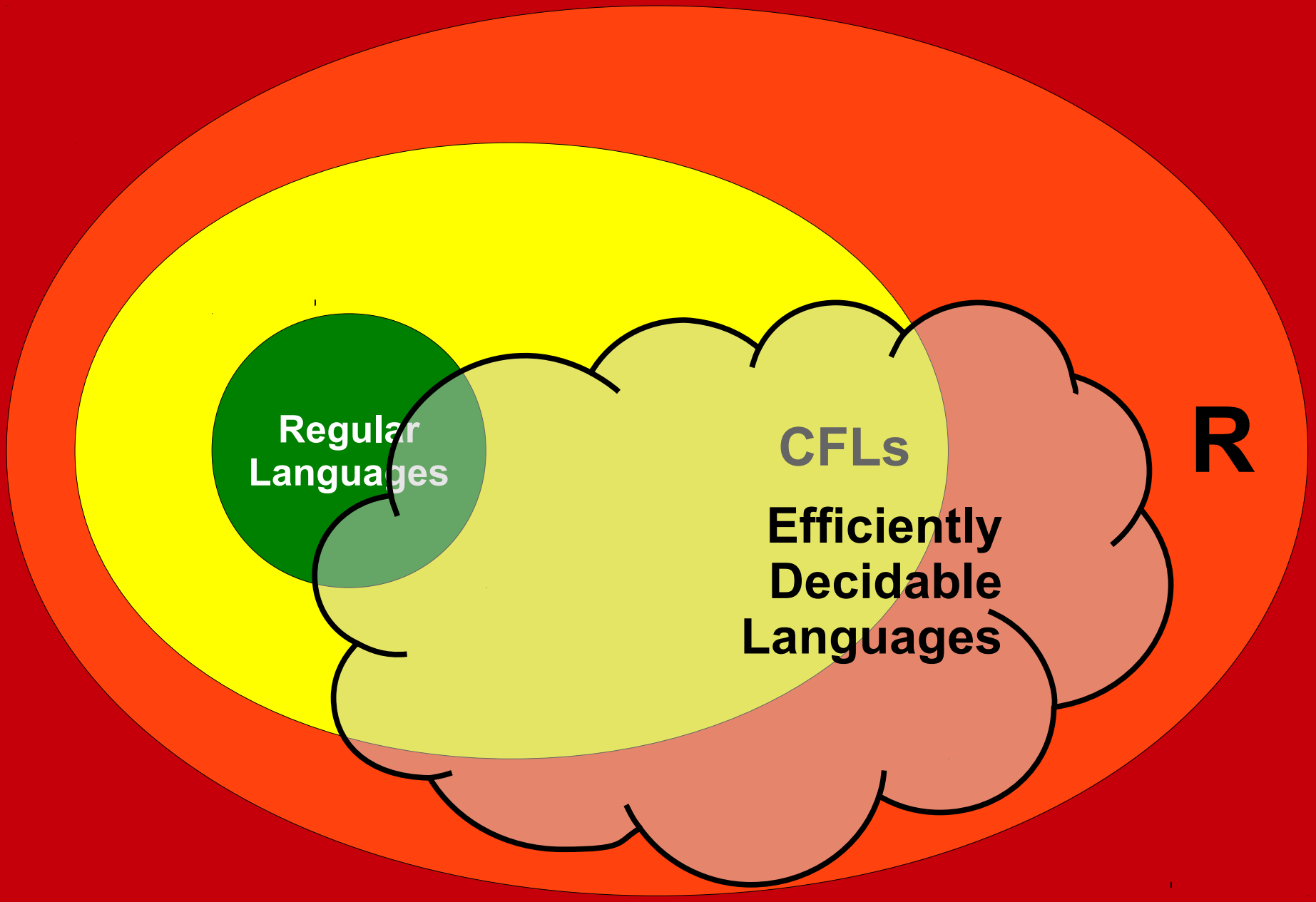
- The ***complexity class* P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

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

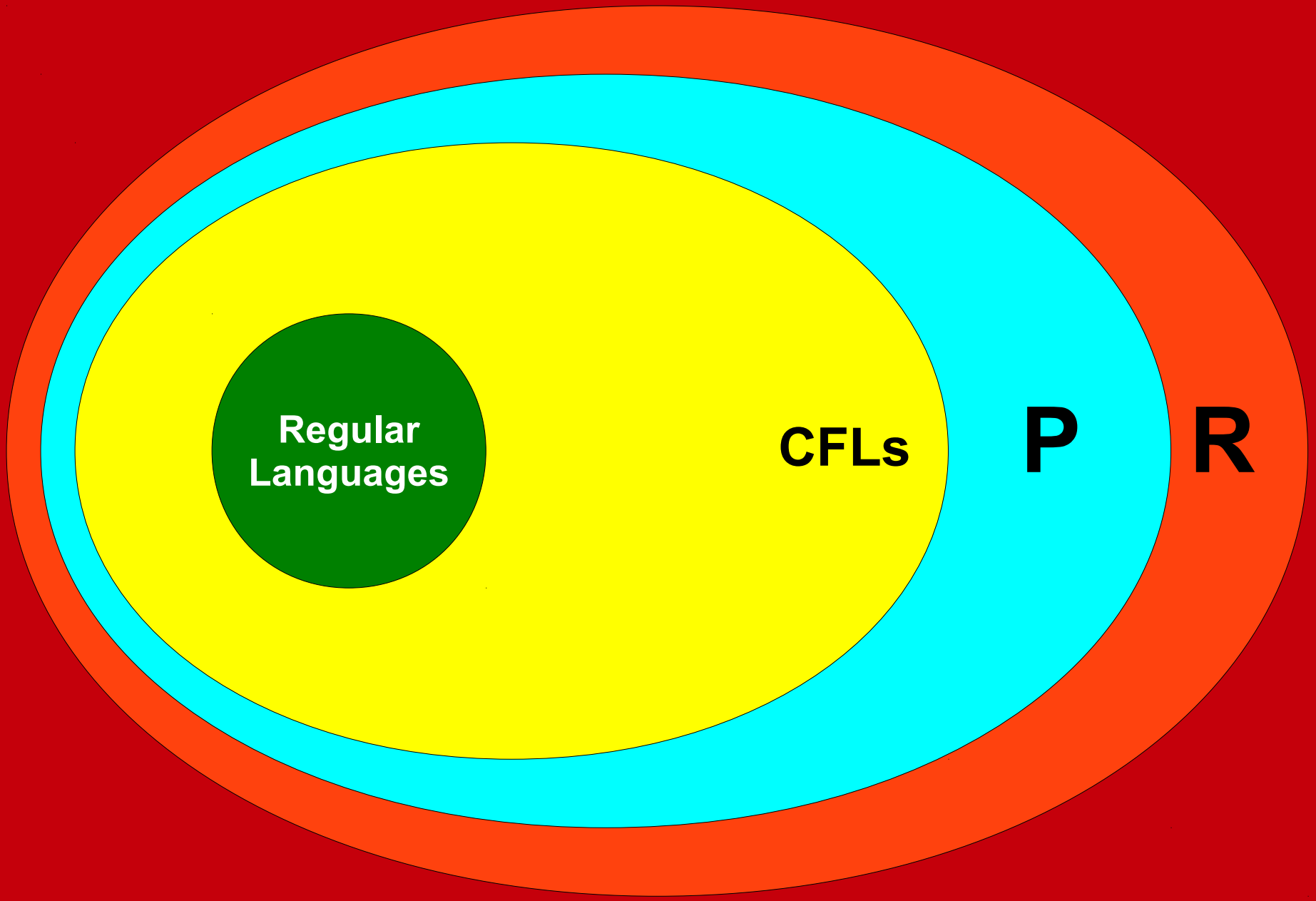
- Assuming the Cobham-Edmonds thesis, a language is in **P** if it can be decided efficiently.

# 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.*)
- Many other problems are in **P**.
  - More on that in a second.




Undecidable Languages




Undecidable Languages

# Problems in **P**

- **Graph connectivity:**

  Given a graph $G$ and nodes $s$ and $t$, is there a path from $s$ to $t$?

- **Primality testing:**

  Given a number $p$, is $p$ prime? (Best known TM for this takes time $O(n^{37})$.)

- **Maximum matching:**

  Given a set of tasks and workers who can perform those tasks, if each worker performs exactly one task, can at least $n$ tasks be performed?

# Problems in **P**

- **Remoteness testing:**

  Given a graph $G$, are all of the nodes in $G$ within distance at most $k$ of one another?

- **Linear programming:**

  Given a linear set of constraints and linear objective function, is the optimal solution at least $n$?

- **Edit distance:**

  Given two strings, can the strings be transformed into one another in at most $n$ single-character edits?

Time-Out for Announcements!

# Problem Sets

- Problem Set Eight was due at the start of today's lecture.
  - Want to use your remaining late days? Feel free to turn it in by next Monday at 3:00PM.
- Problem Set Nine goes out now. It's due on Friday of the last day of class.
  - Explore the limits of the **R** and **RE** languages!
  - Get a feel for complexity theory, what we know, and what we don't.
  - No late submissions accepted – sorry about that!

# Your Questions

“Why don't Stanford Student date?”

Wait, they don't? Most people I knew were dating when they were students here. Crazy kids these days…

“The practice midterm exams and problem sets have been great for studying. Will similar resources be available for the final?”

Oh yeah. We'll release tons of cumulative practice problems, a practice final, more challenge problems, and then probably even more practice problems.

“What ya up to this Thanksgiving? ☺”

Writing practice problems. ☺

Also, visiting family, and helping cook dinner. A recipe you might like…

# Roasted Maple-Glazed Carrots with Thyme and Walnuts

This recipe comes from a cooking class at Sacramento's Food Cooperative. The thyme is the sleeper hit here – it really makes everything come together!

## Ingredients

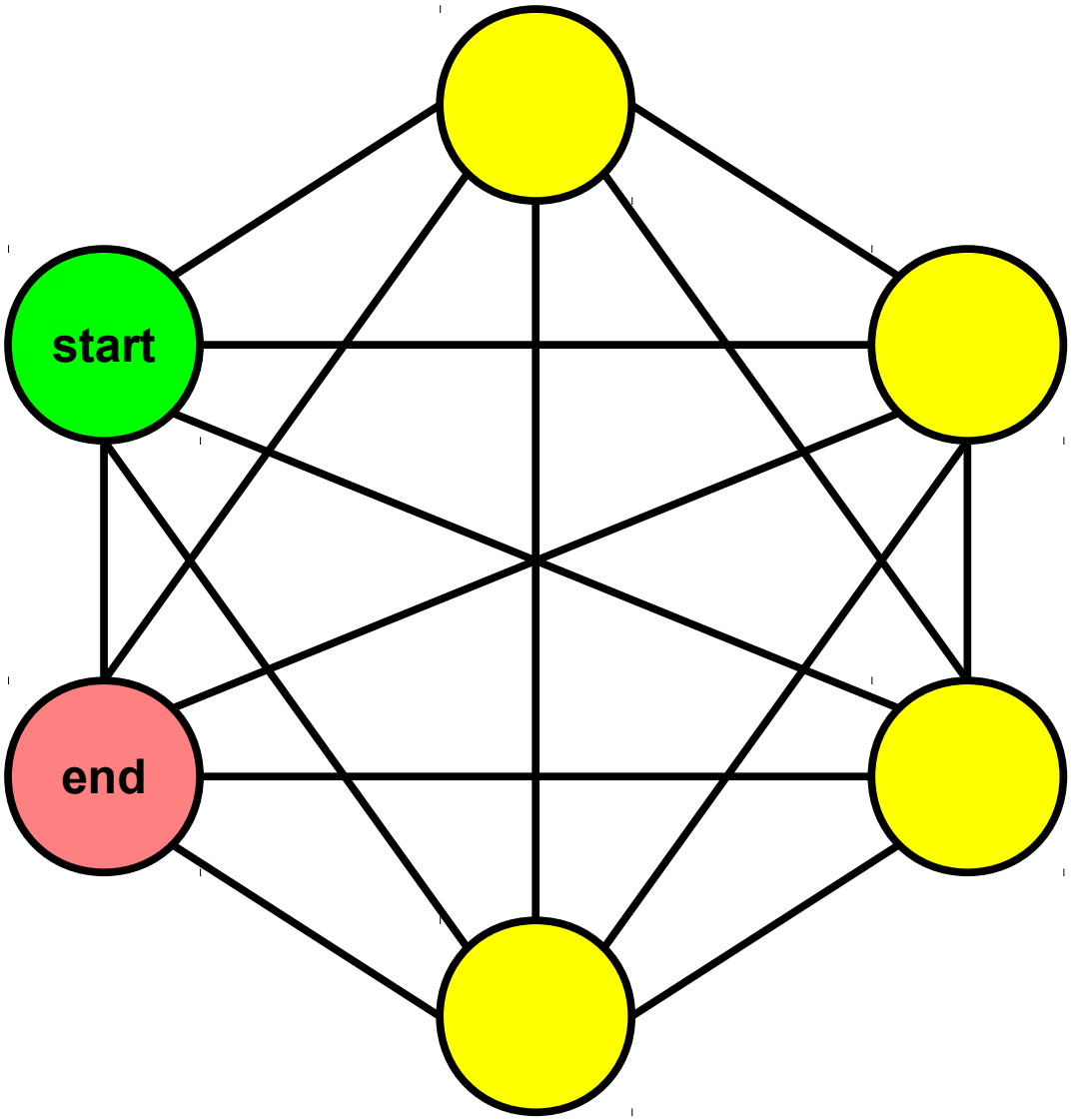
- 2 pounds baby bunch carrots, peeled and sliced from top to bottom.
- 2 tbsp extra virgin olive oil
- Sea salt and freshly ground pepper to taste
- 2 tbsp maple syrup
- 2/3 tbsp melted butter
- 1/4 cup walnuts, toasted and chopped
- 1 tbsp chopped fresh thyme

## Directions

1. Preheat oven to 400°F.
2. Ensure that the carrots are dry. Then, toss with olive oil and season generously with salt and pepper.
3. Spread the carrots out onto a parchment-paper-lined baking sheet, giving the carrots plenty of space. Bake for 20 minutes until slightly brown and remove from oven. (You may want to turn the carrots halfway through the baking to ensure that they brown uniformly.)
4. Combine the maple syrup and melted butter. Pour over the carrots and stir them around to coat well. Increase the heat in the oven to 475°F, then bake for up to 10 minutes to get the glaze to stick to the carrots and turn glossy. Be careful - this can burn if you don't keep an eye on it.
5. Sprinkle with walnuts and fresh thyme and toss to combine. Season with salt to taste. Serve warm or at room temperature.

Back to CS103!

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




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


How many subsets of this set are there?

# An Interesting Observation

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

NP

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 |

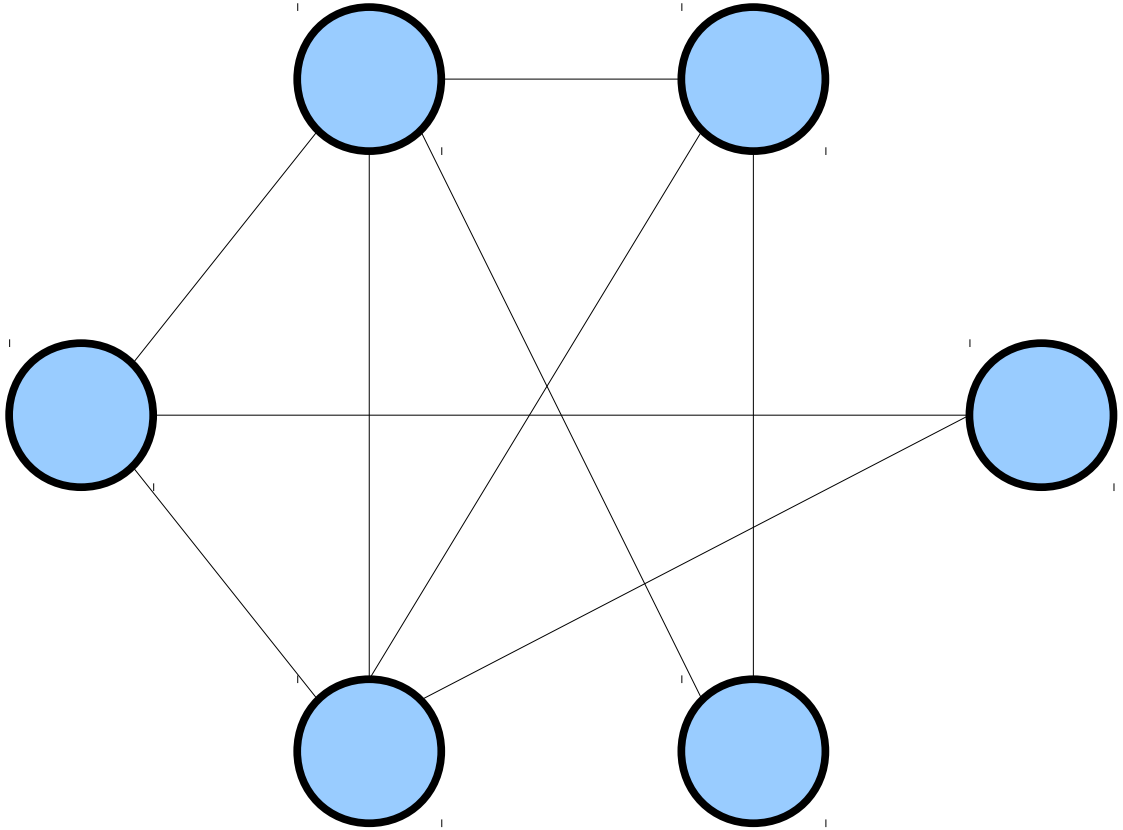
Is there an ascending subsequence of length at least 7?

# Verifiers – Again

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

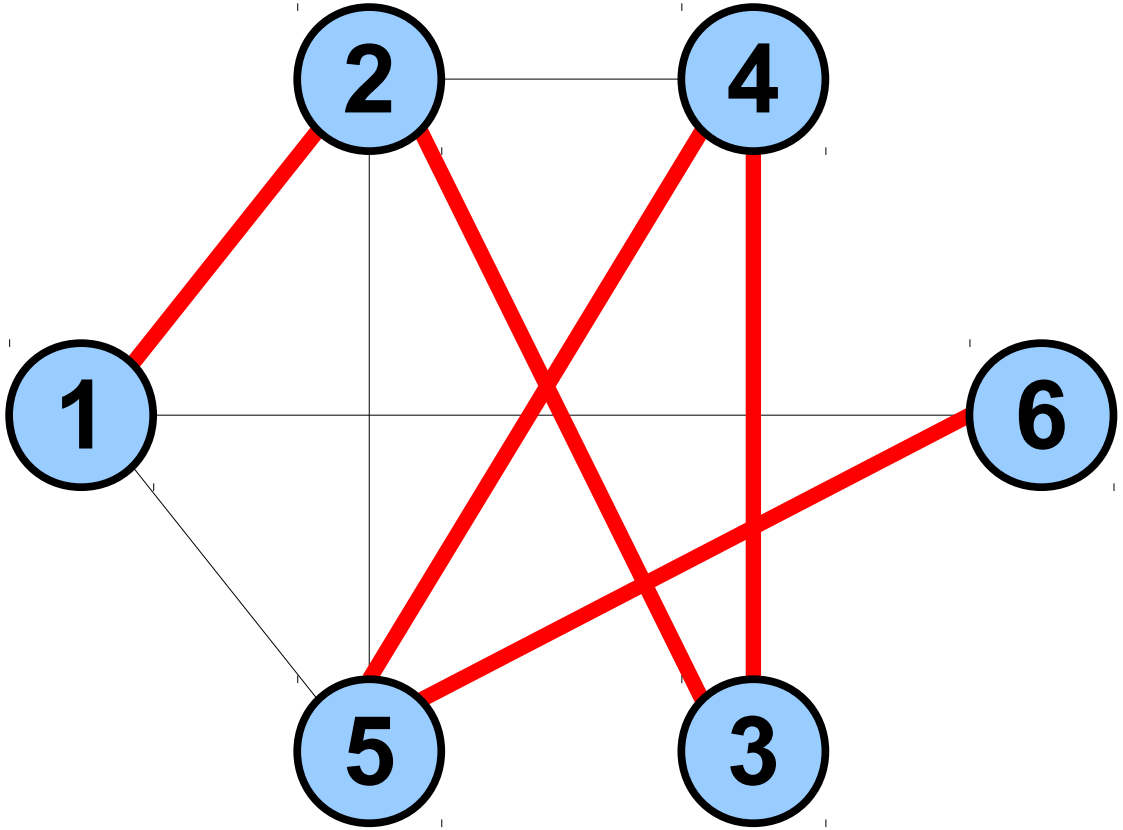
Is there an ascending subsequence of length at least 7?

# Verifiers – Again



Is there 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.

# Sample Problems in **NP**

- All the preceding problems are in **NP**:
  - Is a Sudoku solvable?
  - Does a graph $G$ have a Hamiltonian path?
  - Is there an increasing subsequence of length at least $k$?
- Plus *a lot* of others:
  - Given a graph $G$, is $G$ 3-colorable?
    - There are $3^n$ possible colorings of the nodes in an $n$-node graph, but if someone gives you a valid coloring, it's easy to check.
  - Given a graph $G$, is $G$ Hamiltonian?
    - There are $n!$ possible orderings of the nodes in an $n$-node graph, but if someone gives you a correct ordering of the nodes to form a Hamiltonian path, it's easy to check.

And now…

The

***Most Important Question***

in

***Theoretical Computer Science***

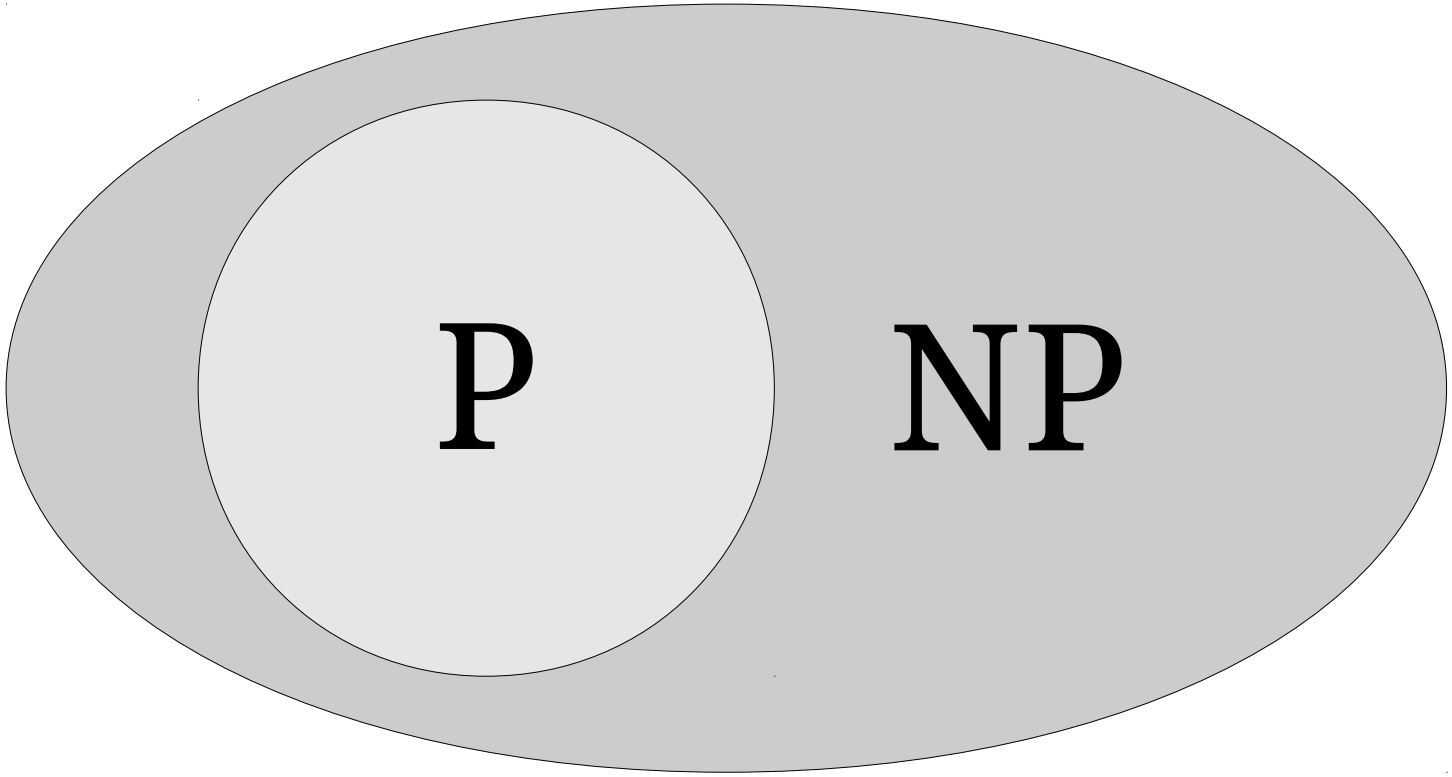
What is the connection between **P** and **NP**?

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

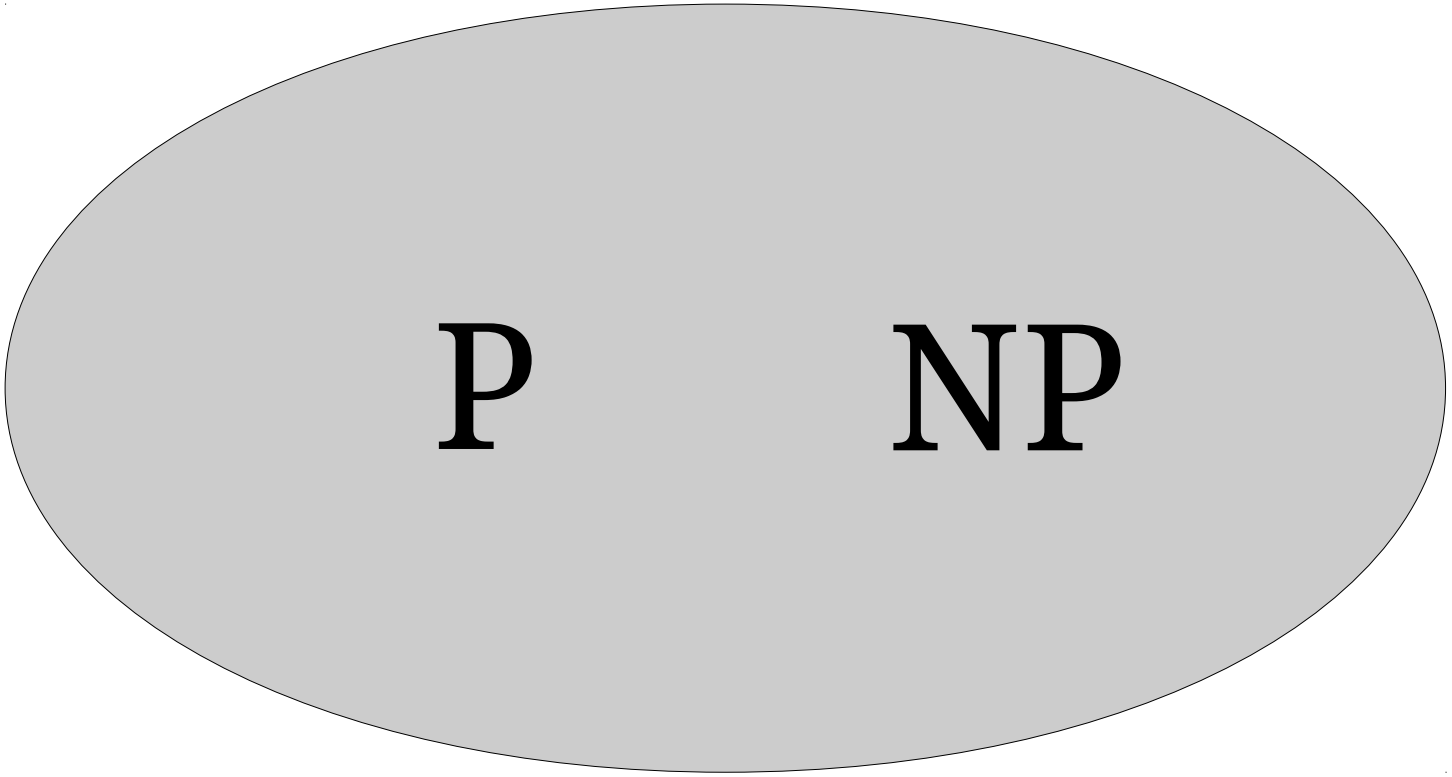
$$\mathbf{NP} = \{\ L \mid \text{There is a polynomial-time verifier for } L\ \}$$

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

# Which Picture is Correct?



# Which Picture is Correct?



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

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

- The $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question is the most important question in theoretical computer science.
- With the verifier definition of **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 ≠ NP**, ***none*** of these problems have efficient solutions.

# Why This Matters

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

# What We Know

- Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ has proven ***extremely difficult***.
- In the past 43 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.ing.puc.cl/~jabaier/iic2212/poll-1.pdf

# The Million-Dollar Question

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

# The Million-Dollar Question


The Clay Mathematics Institute has offered a **$1,000,000 prize** to anyone who proves or disproves **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**?

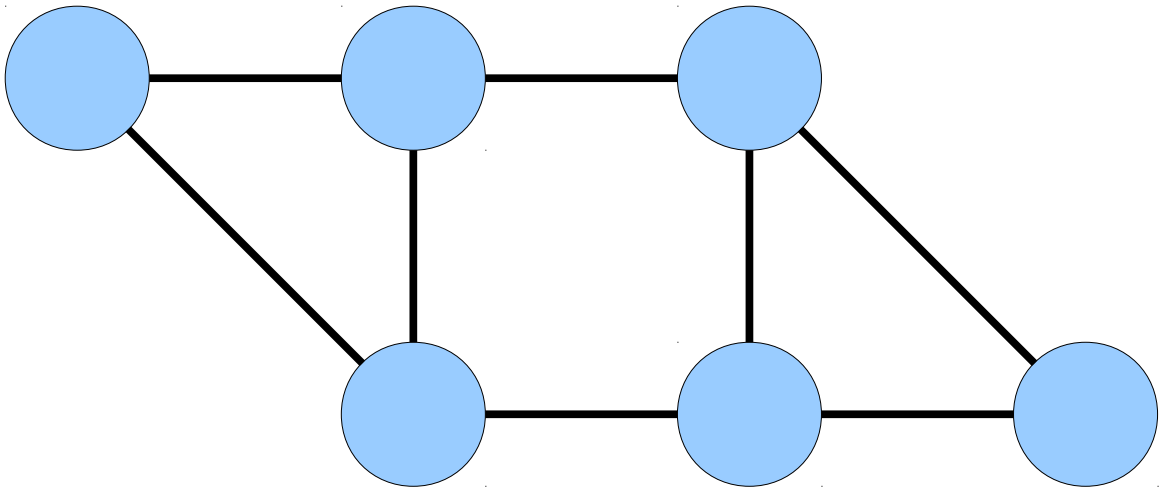
# Reducibility

# 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.

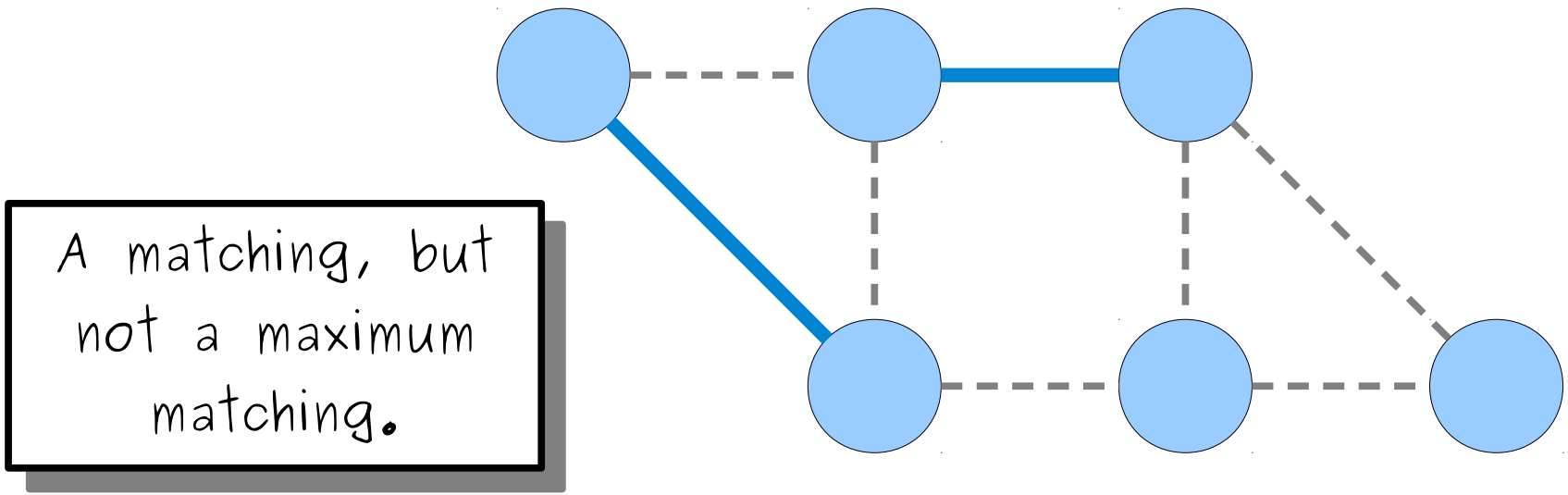
# Maximum Matching

- 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.



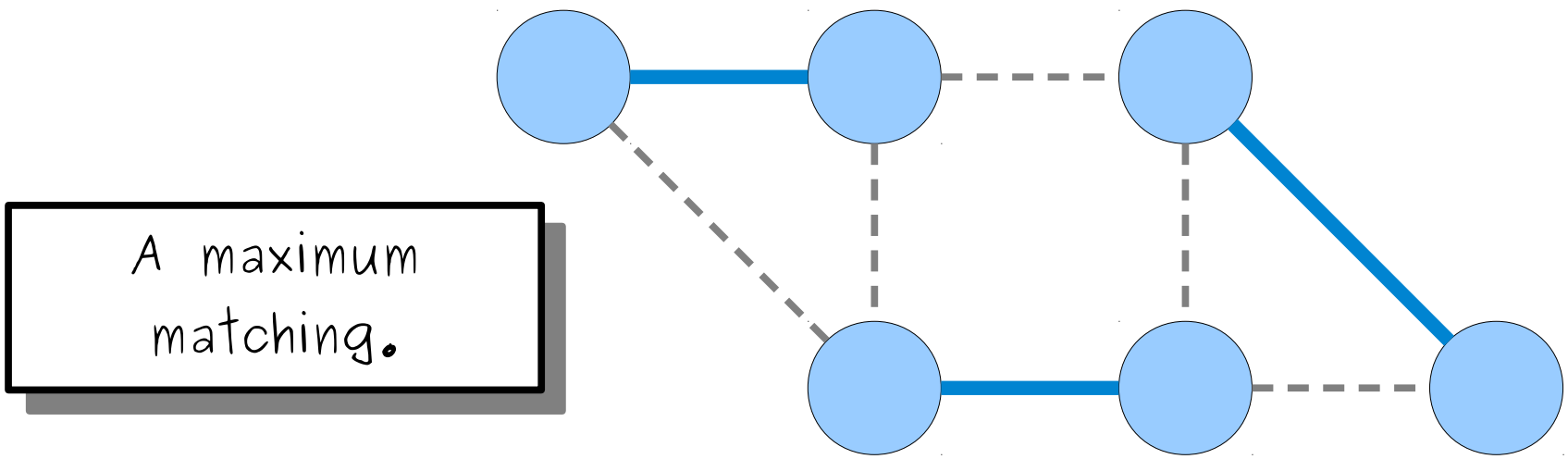
# Maximum Matching

- 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.



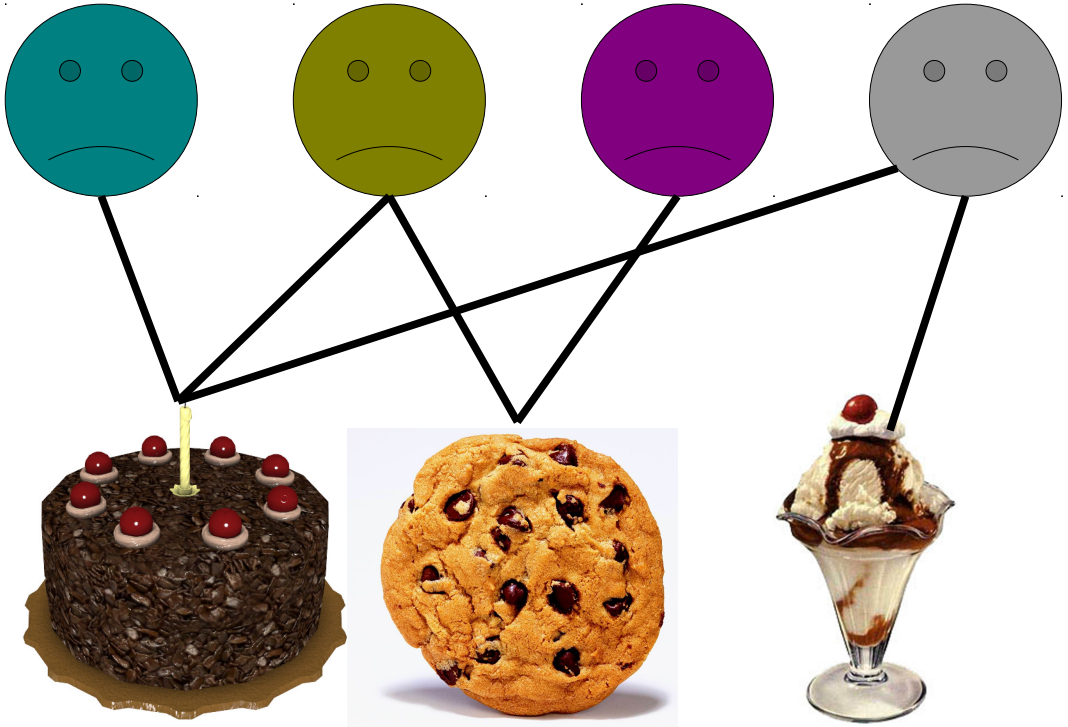
# Maximum Matching

- 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.



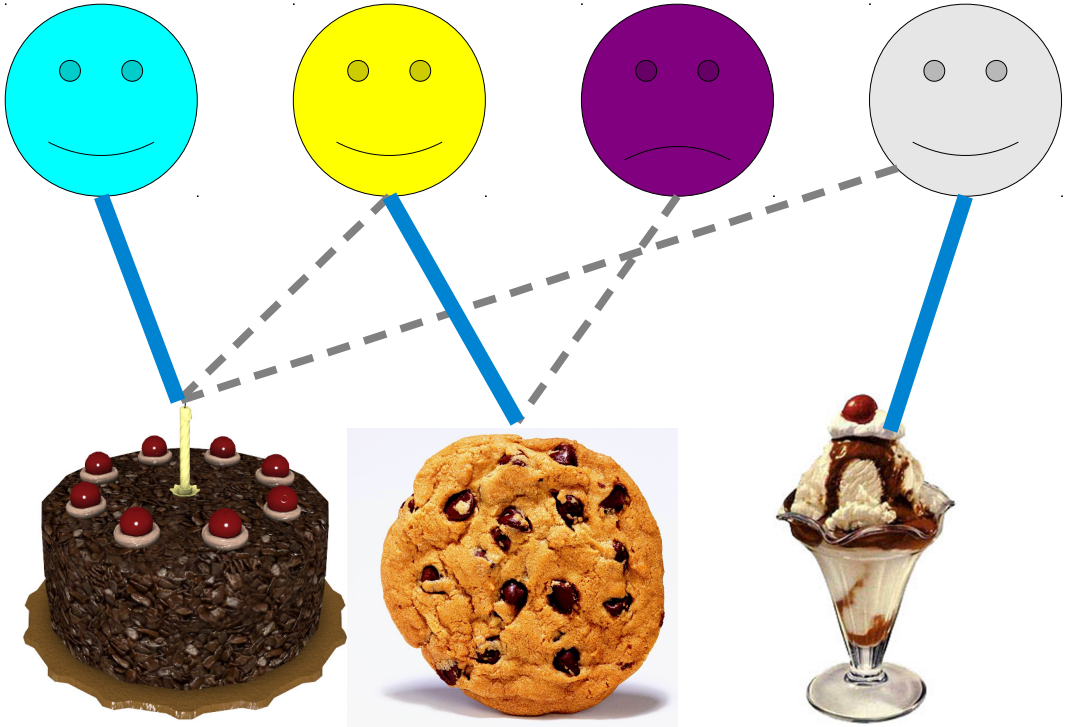
# Maximum Matching

- 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.



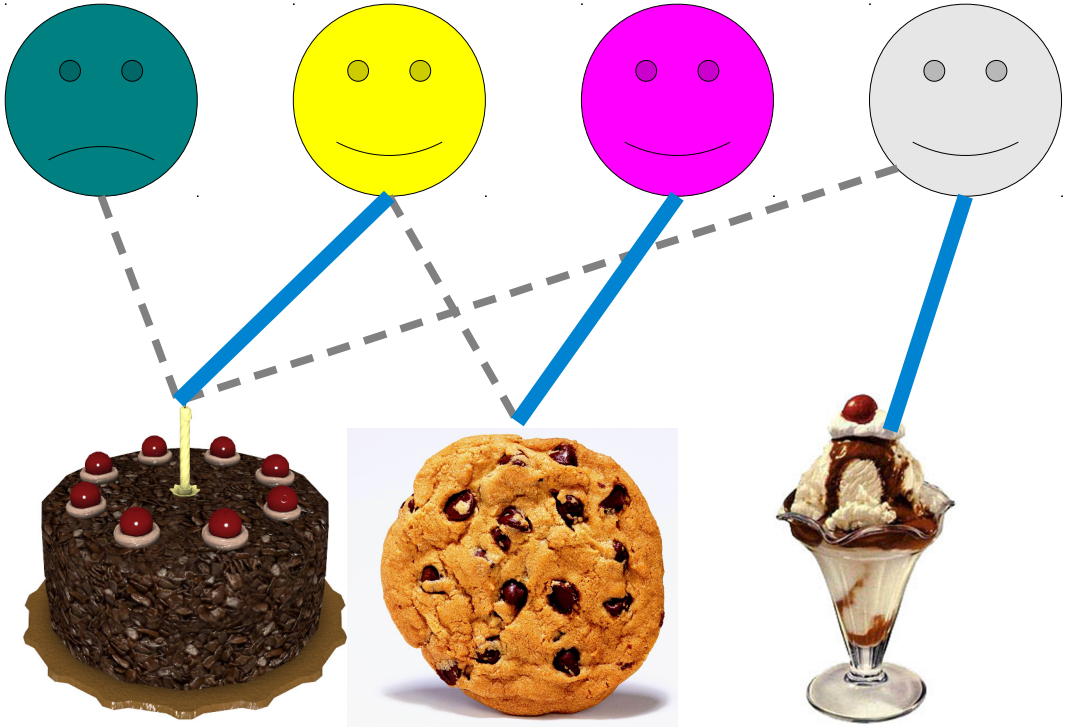
# Maximum Matching

- 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

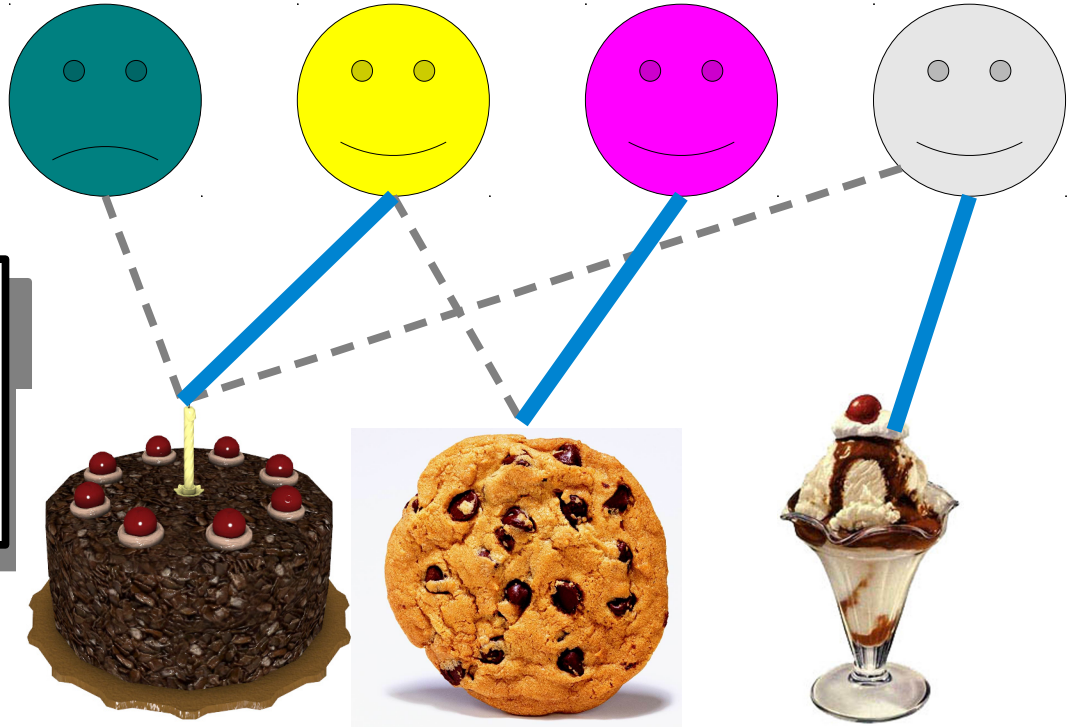
- Given an undirected graph $G$, a ***matching*** in $G$ is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



# 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.

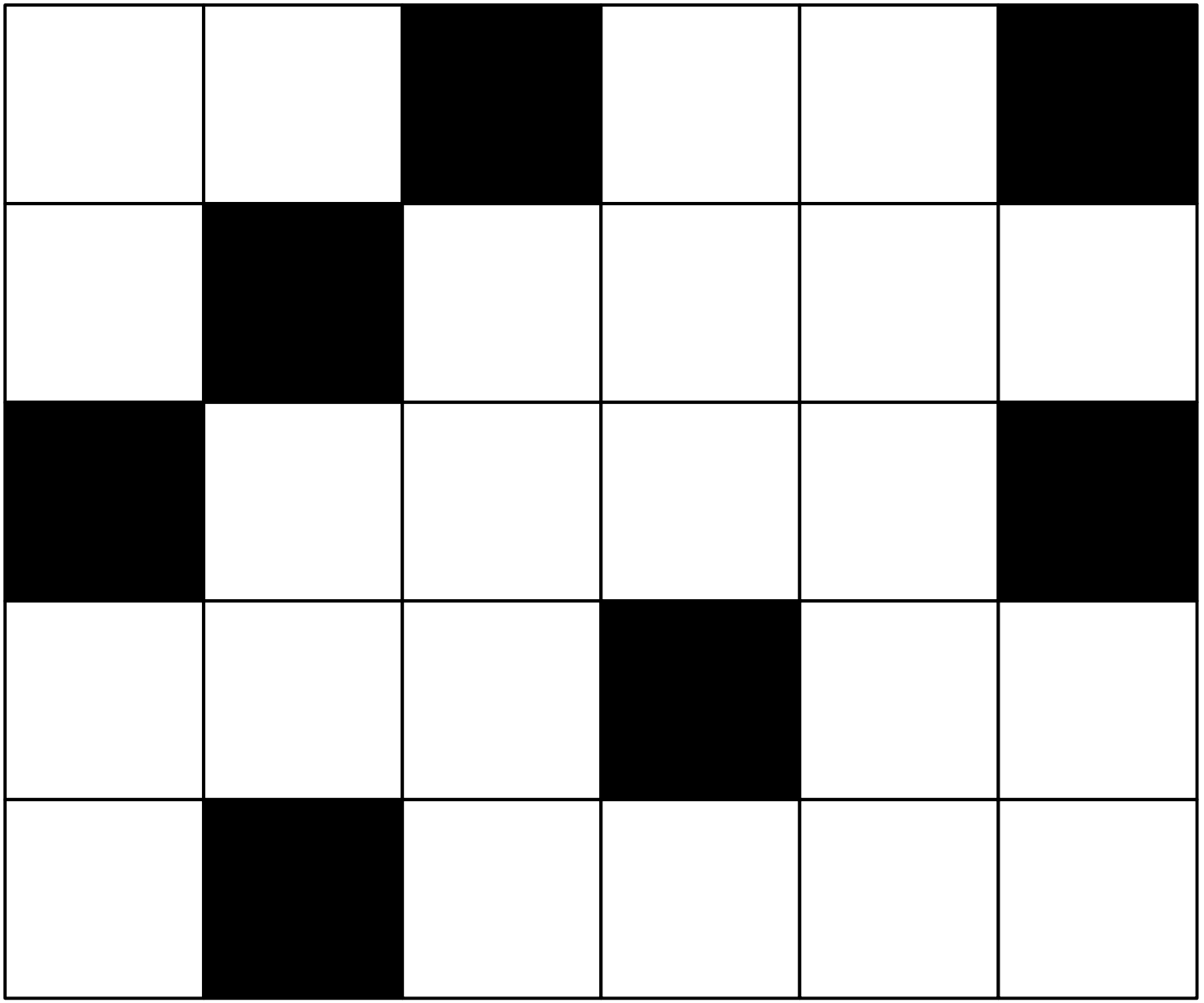
Maximum matchings are not necessarily unique.



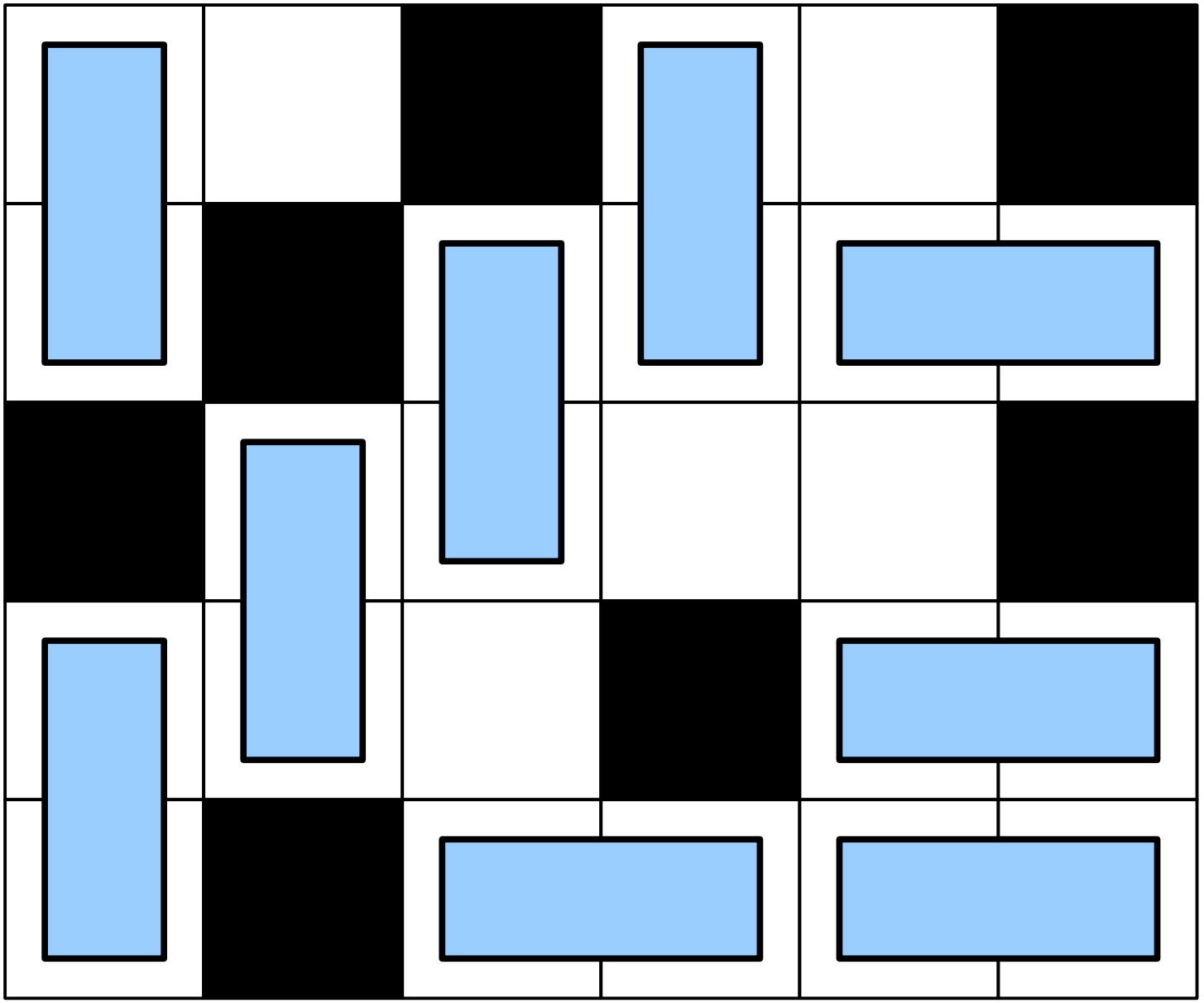
# Maximum Matching

- 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?

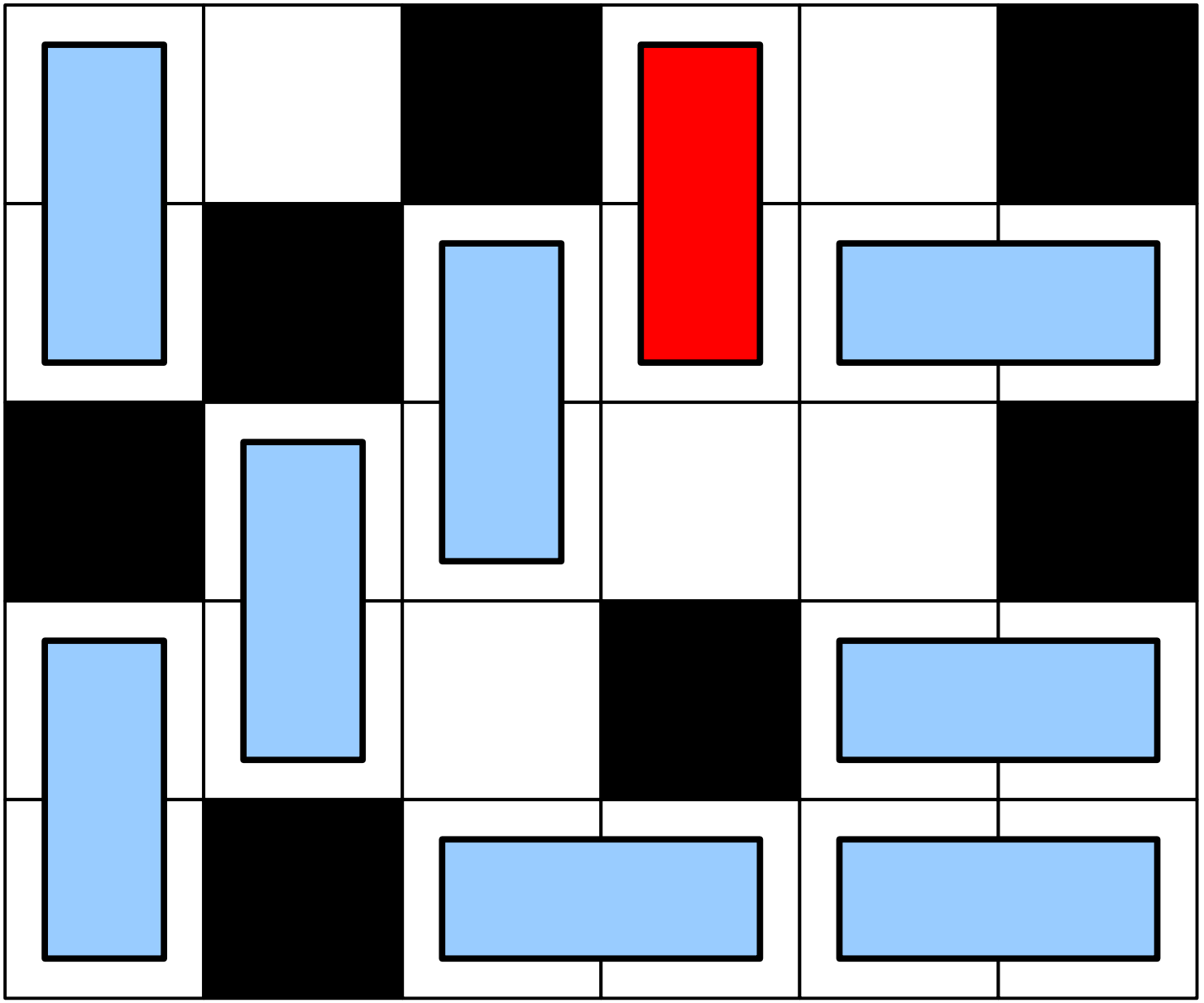
# Domino Tiling



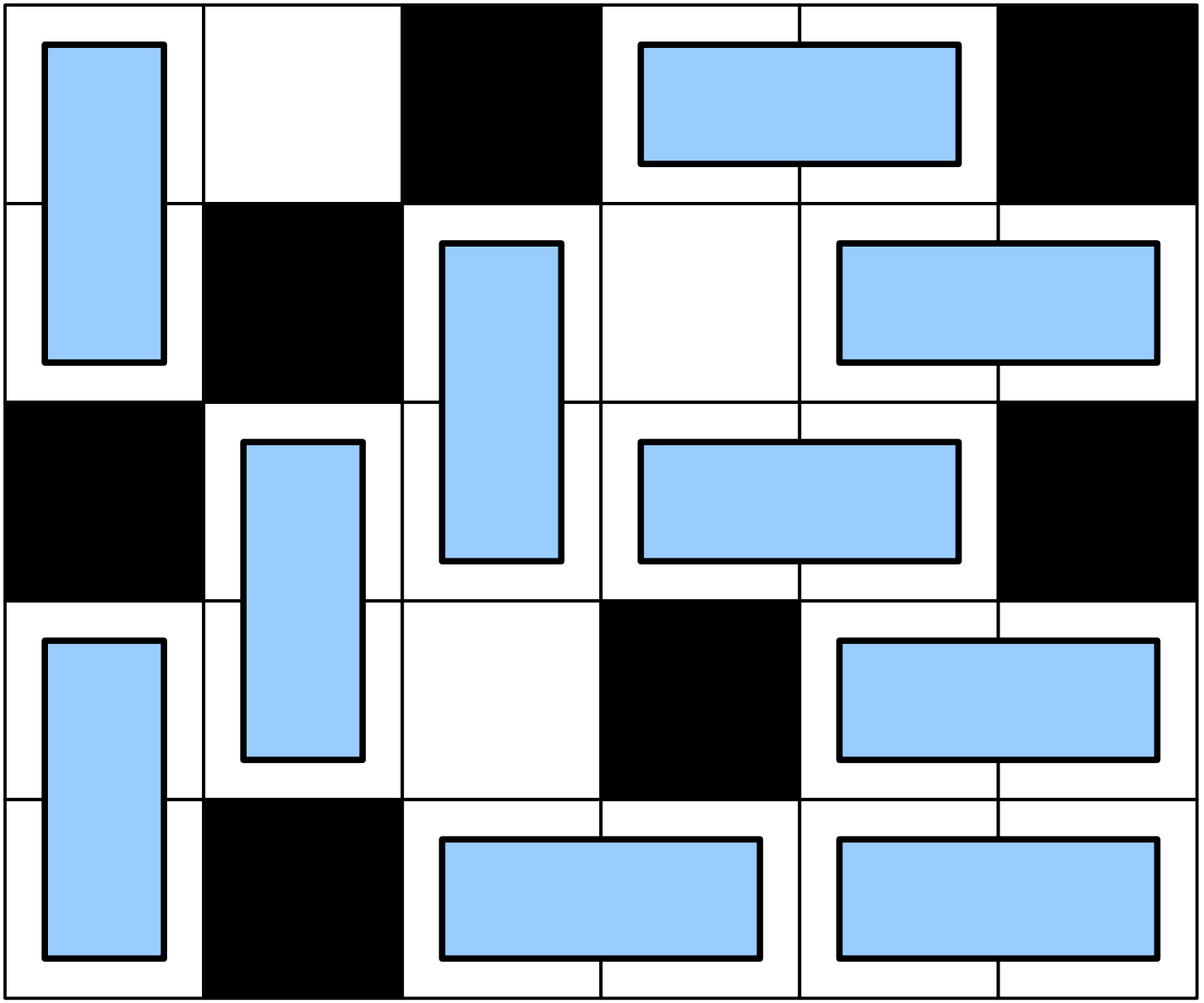
# Domino Tiling



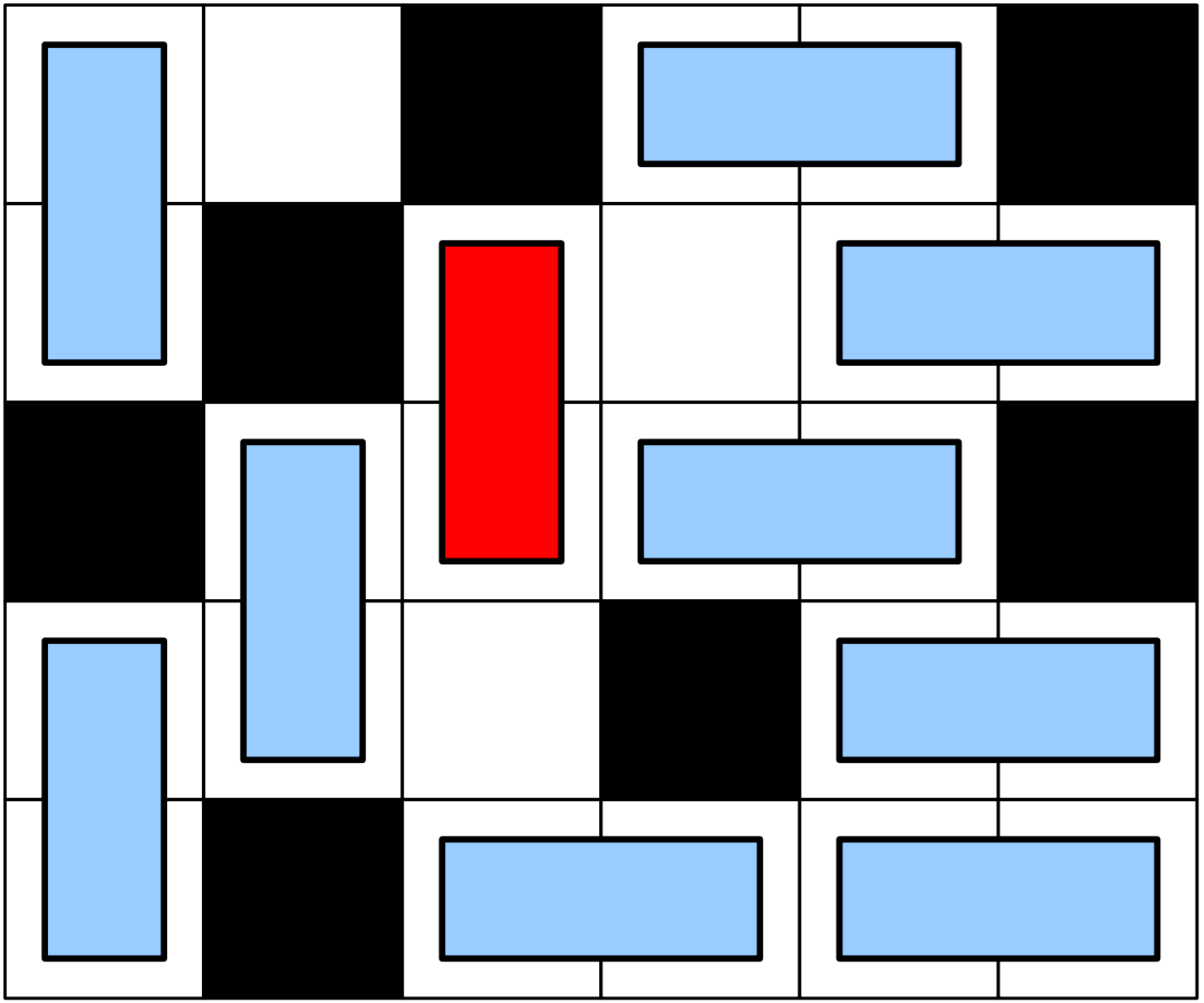
# Domino Tiling



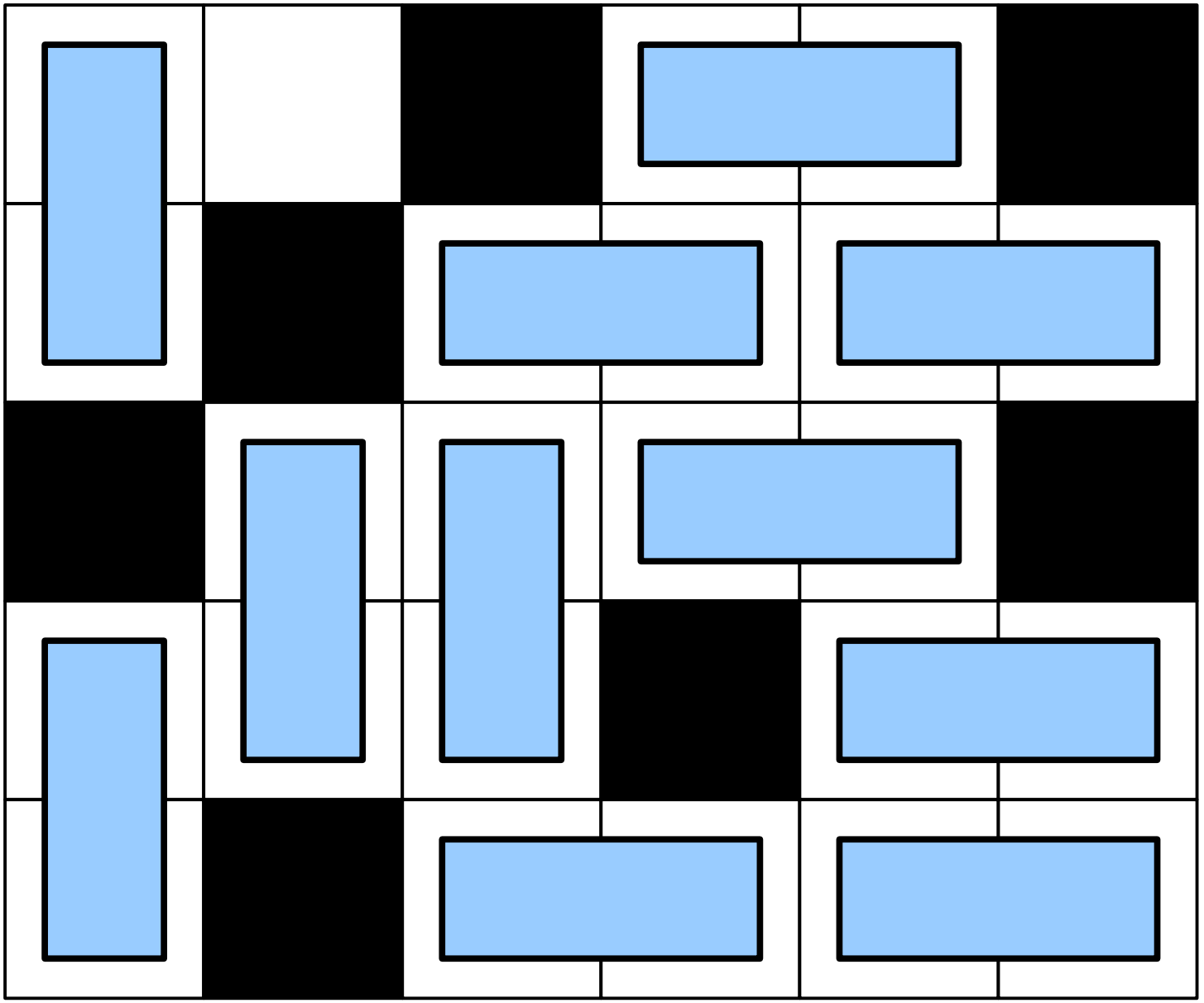
# Domino Tiling



# Domino Tiling



# Domino Tiling



# A Domino Tiling Reduction

- Let *MATCHING* be the language defined as follows:

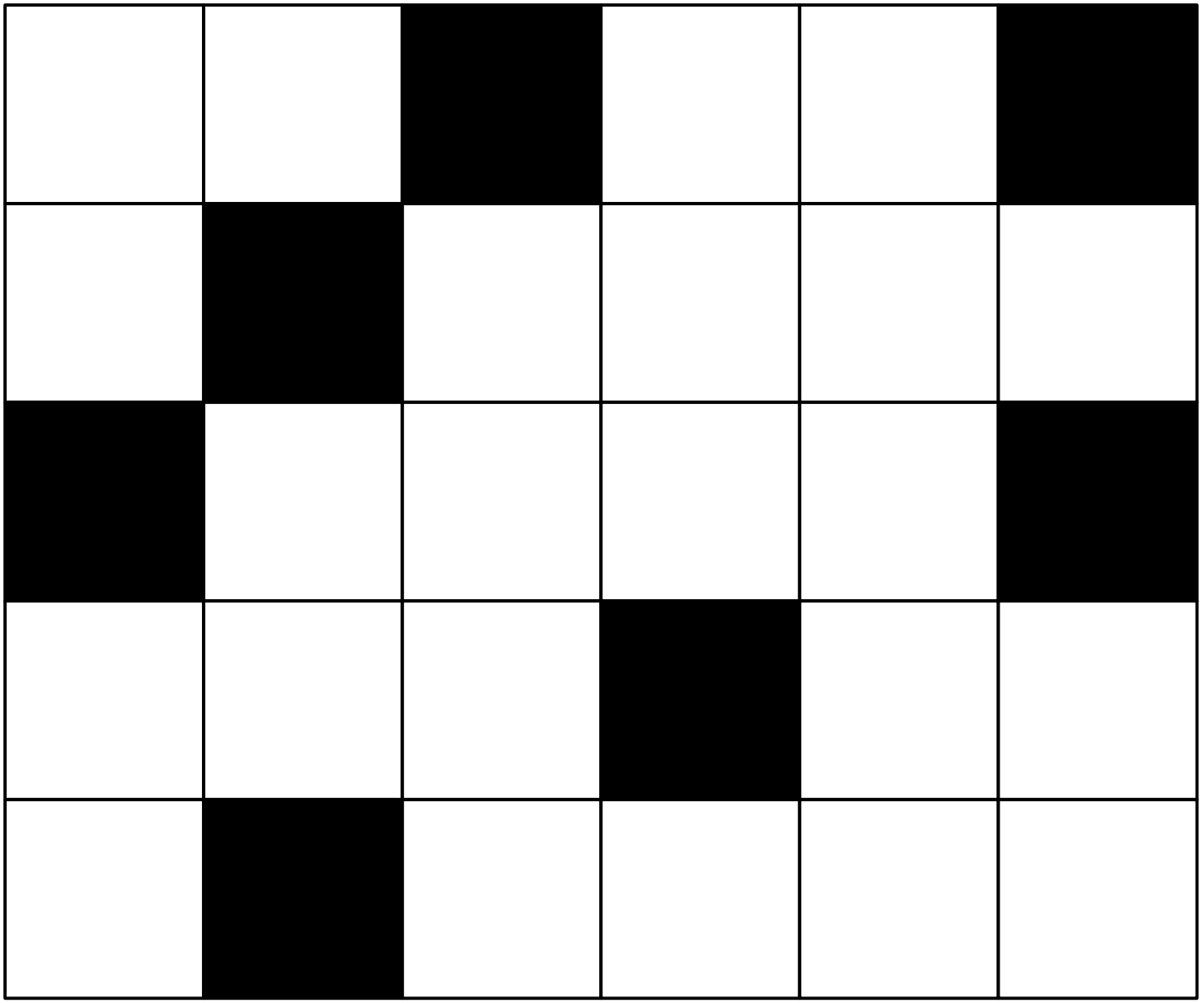
  $MATCHING = \{ \langle G, k \rangle \mid G$ is an undirected graph with a matching of size at least $k$ $\}$

- ***Theorem (Edmonds)***: $MATCHING \in \mathbf{P}$.
- Let *DOMINO* be this language:

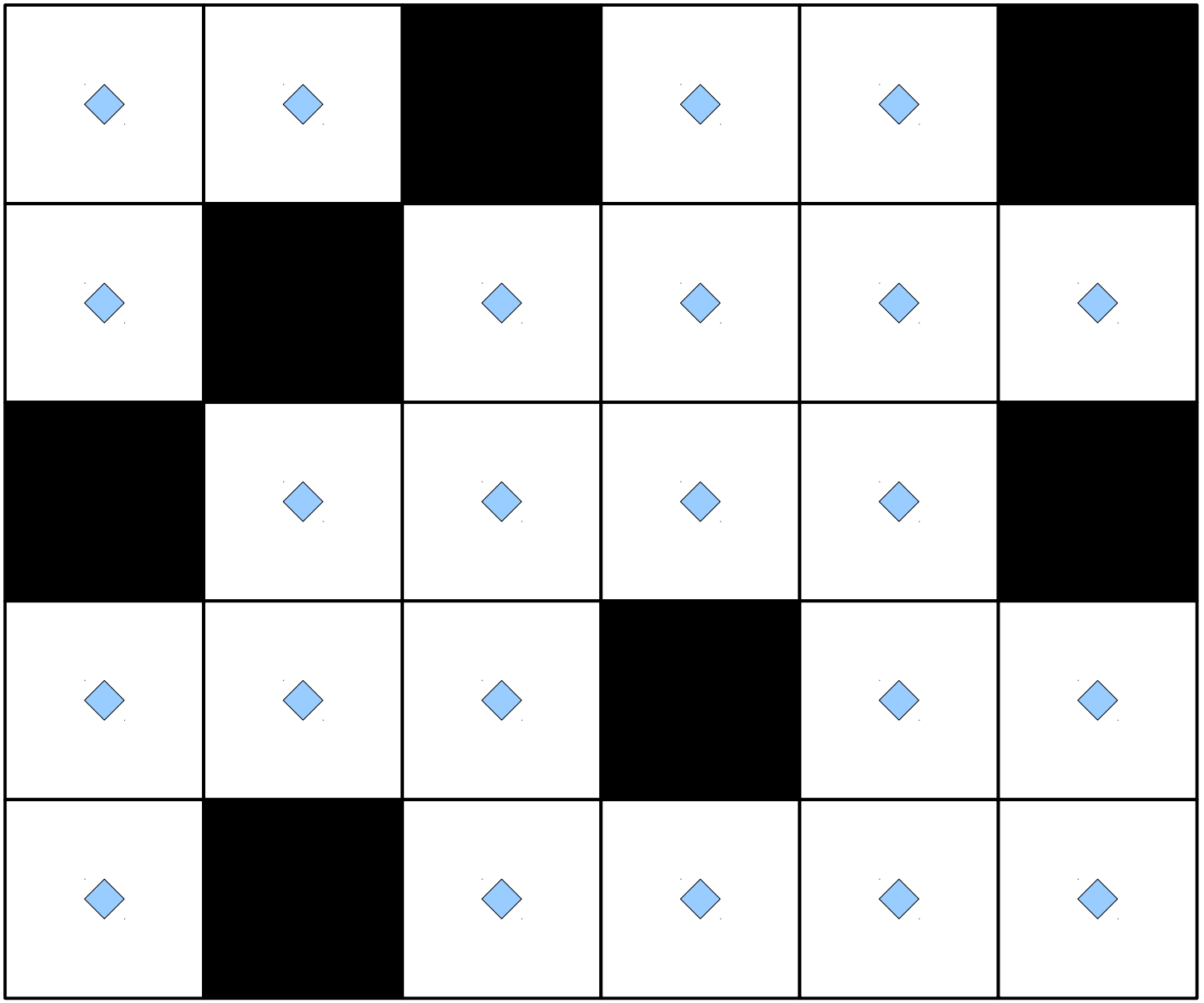
  $DOMINO = \{ \langle D, k \rangle \mid D$ is a grid and $k$ nonoverlapping dominoes can be placed on $D$. $\}$

- We'll use the fact that $MATCHING \in \mathbf{P}$ to prove that $DOMINO \in \mathbf{P}$.

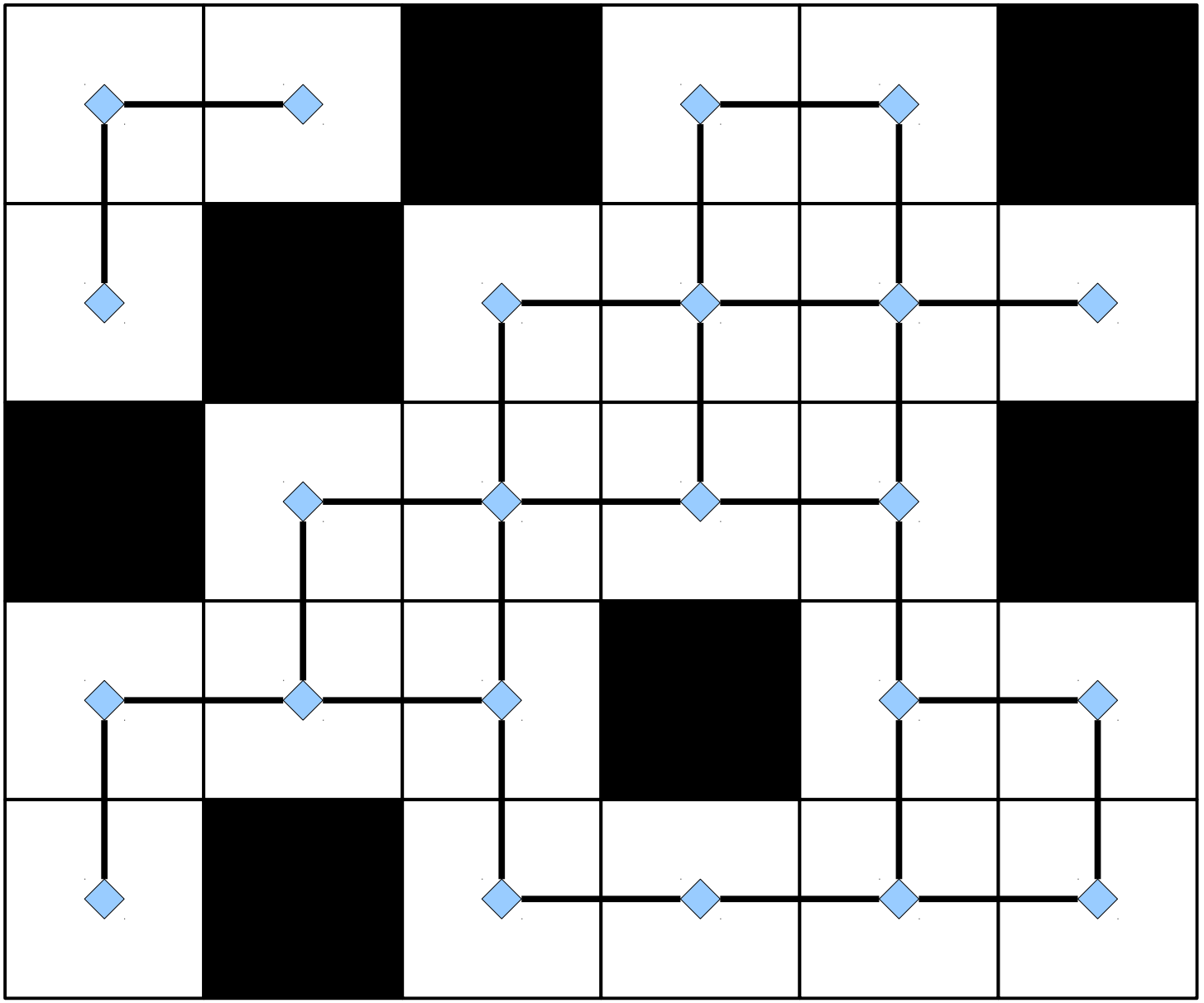
# Solving Domino Tiling



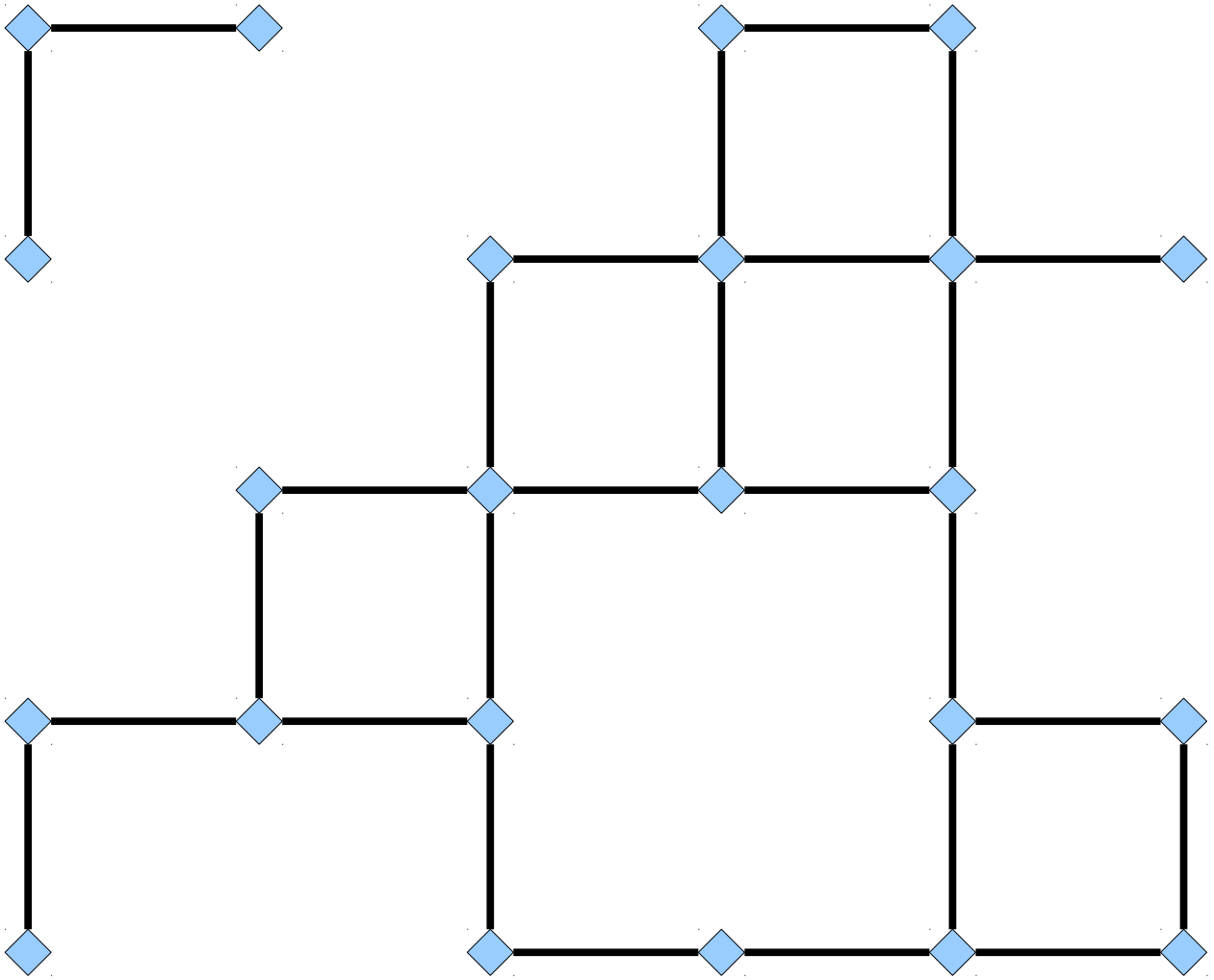
# Solving Domino Tiling



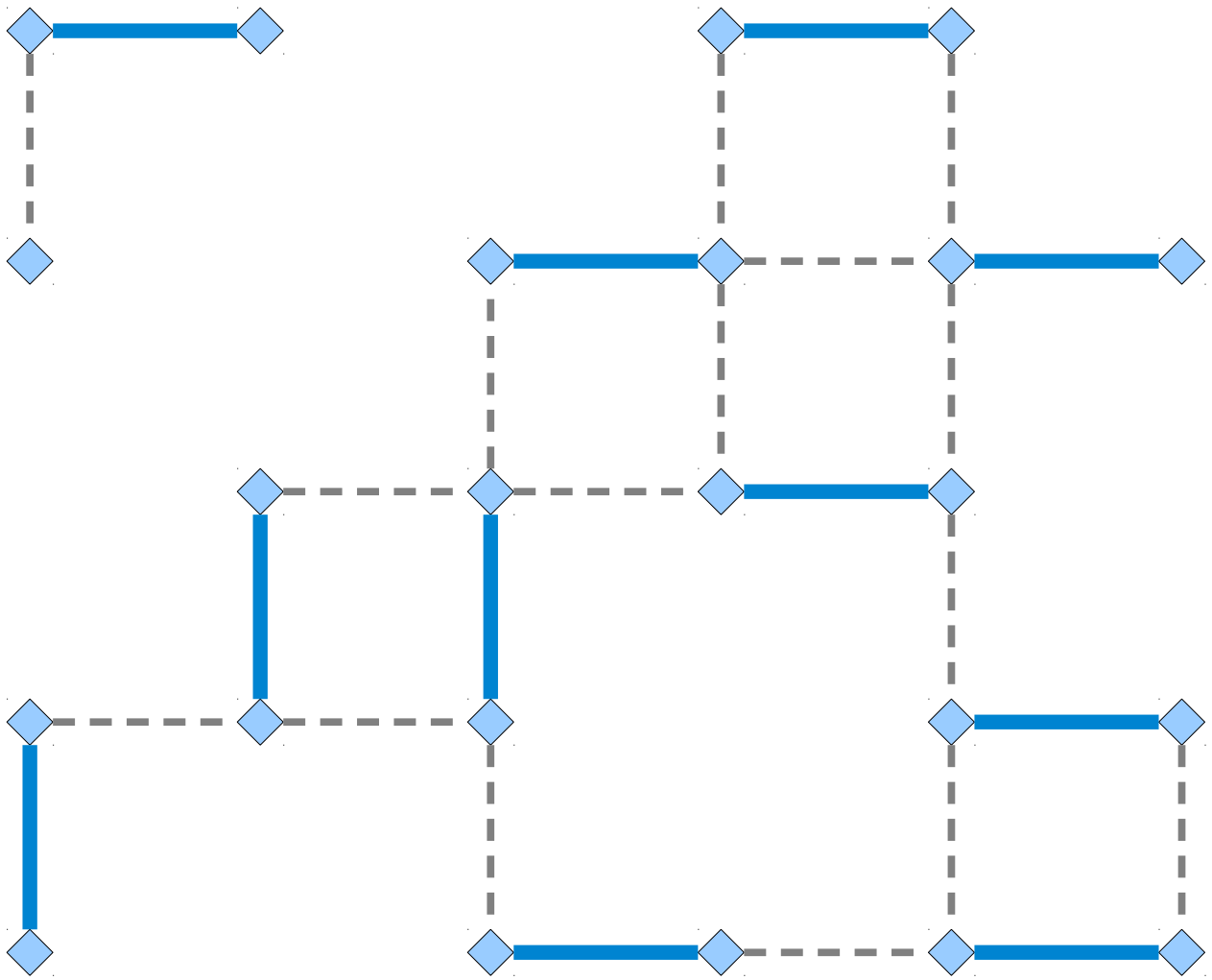
# Solving Domino Tiling



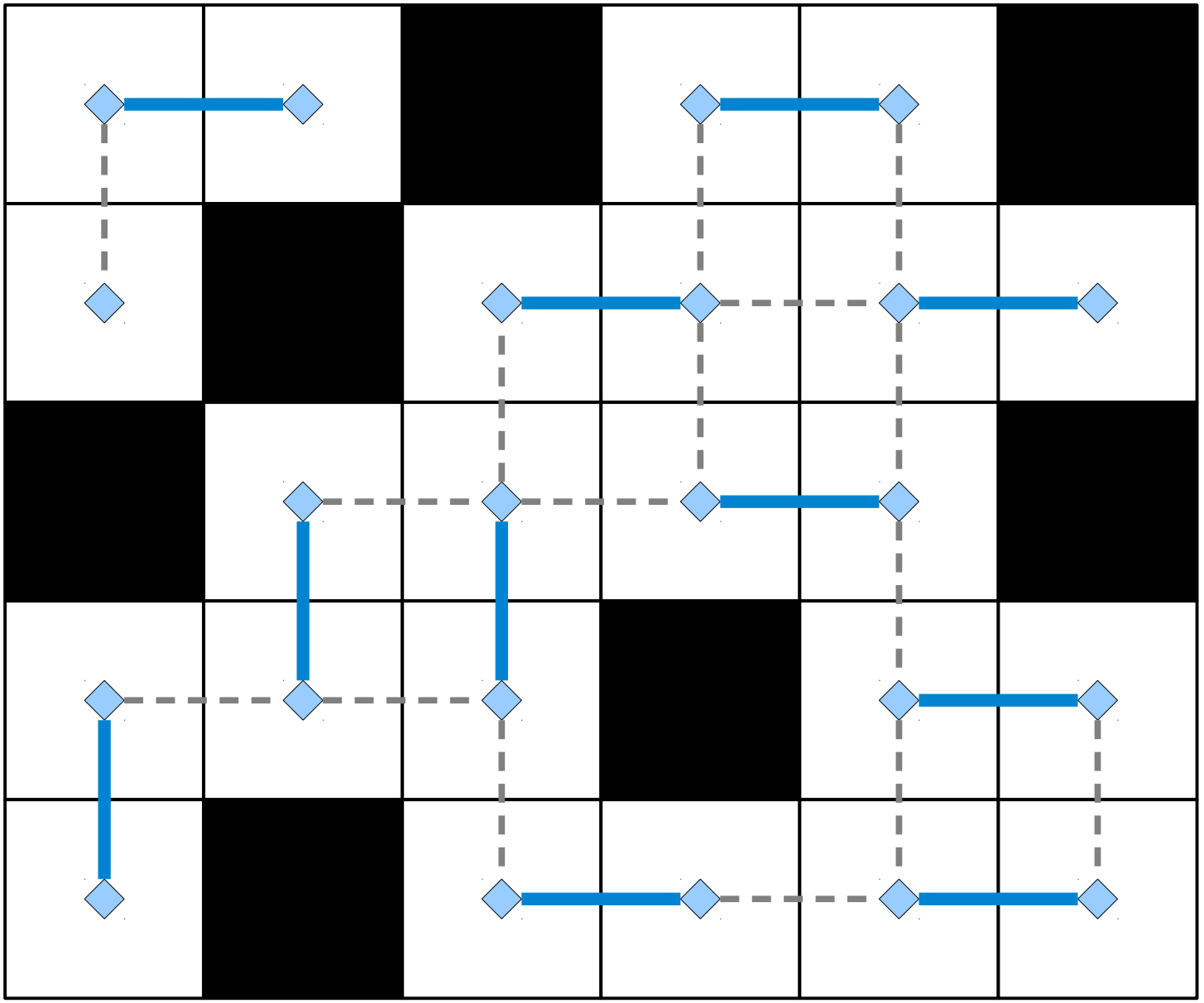
# Solving Domino Tiling



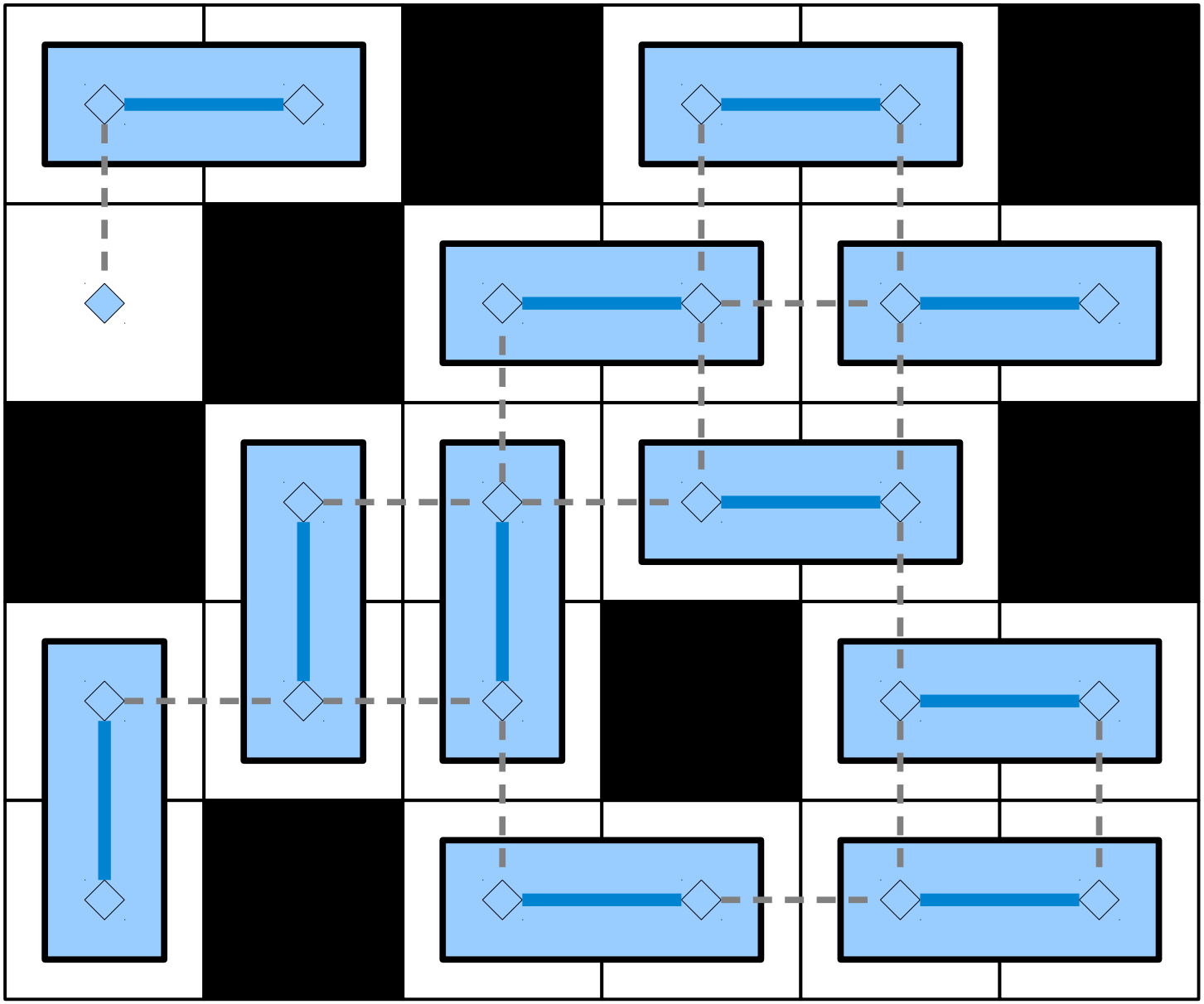
# Solving Domino Tiling



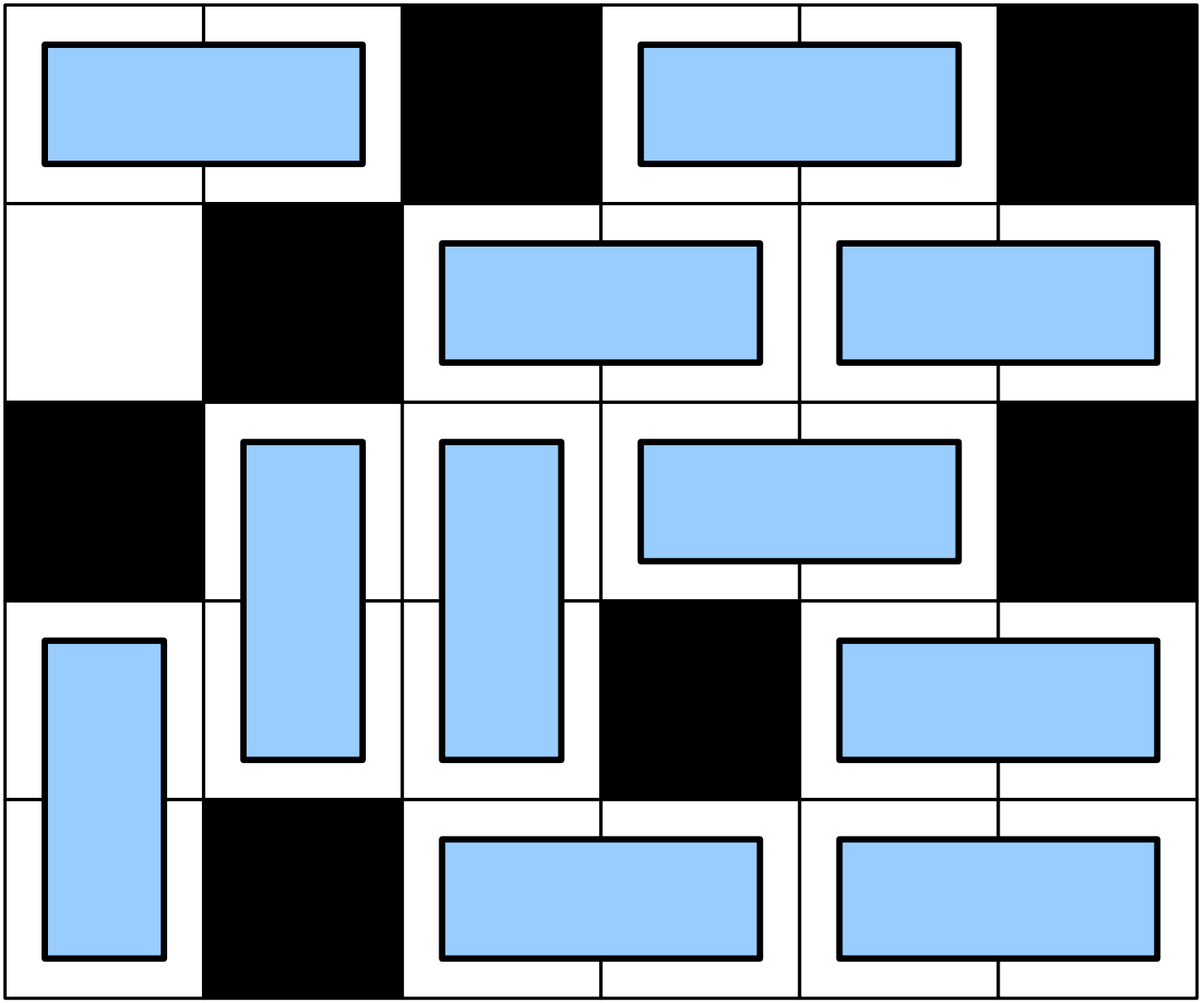
# Solving Domino Tiling



# Solving Domino Tiling



# Solving Domino Tiling



# In Pseudocode

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

# Another Example

# Boggle Spelling

Given a set of Boggle cubes and a target word or phrase, is it possible to arrange the cubes so that you spell out that word or phrase?

# Boggle Spelling

**C U B E**

# Boggle Spelling

**C U B E**

| | | |
|---|---|---|
| | E | |
| O | T | A |
| | I | |
| | N | |

| | | |
|---|---|---|
| | S | |
| H | R | D |
| | L | |
| | U | |

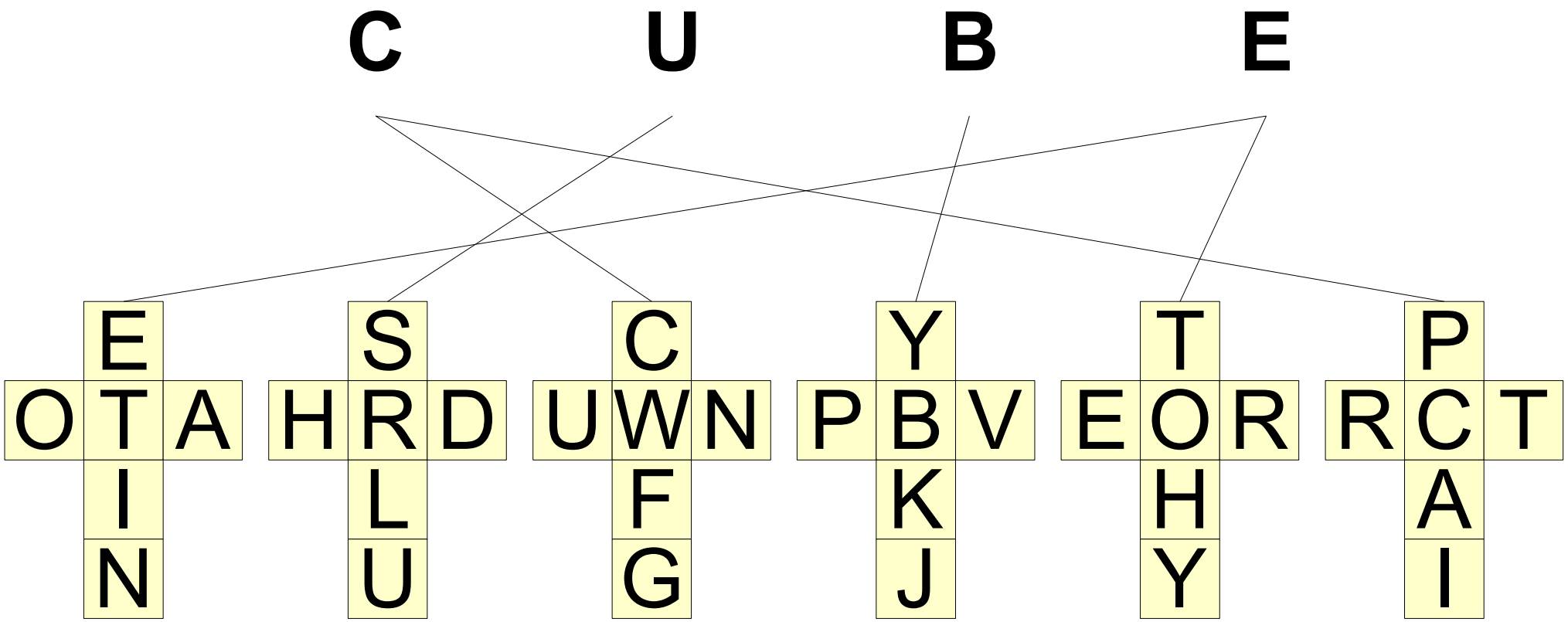
| | | |
|---|---|---|
| | C | |
| U | W | N |
| | F | |
| | G | |

| | | |
|---|---|---|
| | Y | |
| P | B | V |
| | K | |
| | J | |

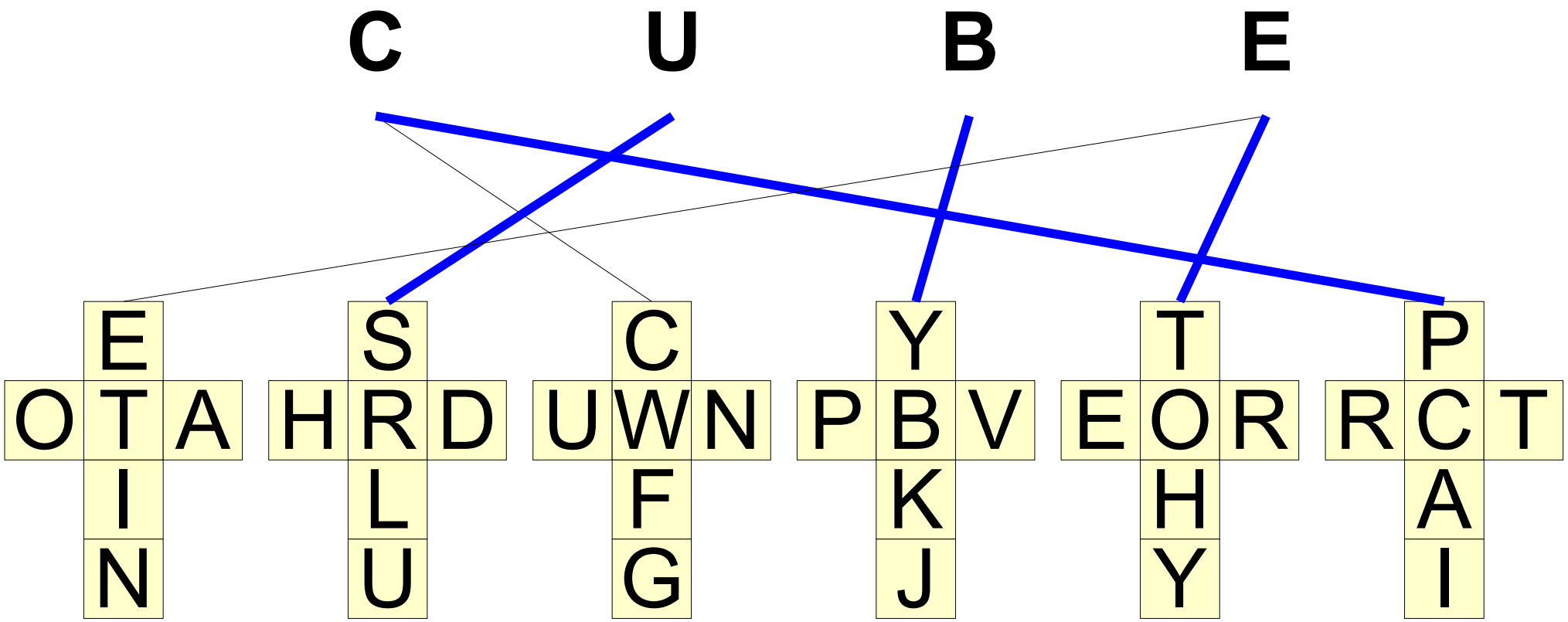
| | | |
|---|---|---|
| | T | |
| E | O | R |
| | H | |
| | Y | |

| | | |
|---|---|---|
| | P | |
| R | C | T |
| | A | |
| | I | |

# Boggle Spelling



# Boggle Spelling



# Boggle Spelling

**C** **F** **G** **S**

| | | |
|---|---|---|
| | E | |
| O | T | A |
| | I | |
| | N | |

| | | |
|---|---|---|
| | S | |
| H | R | D |
| | L | |
| | U | |

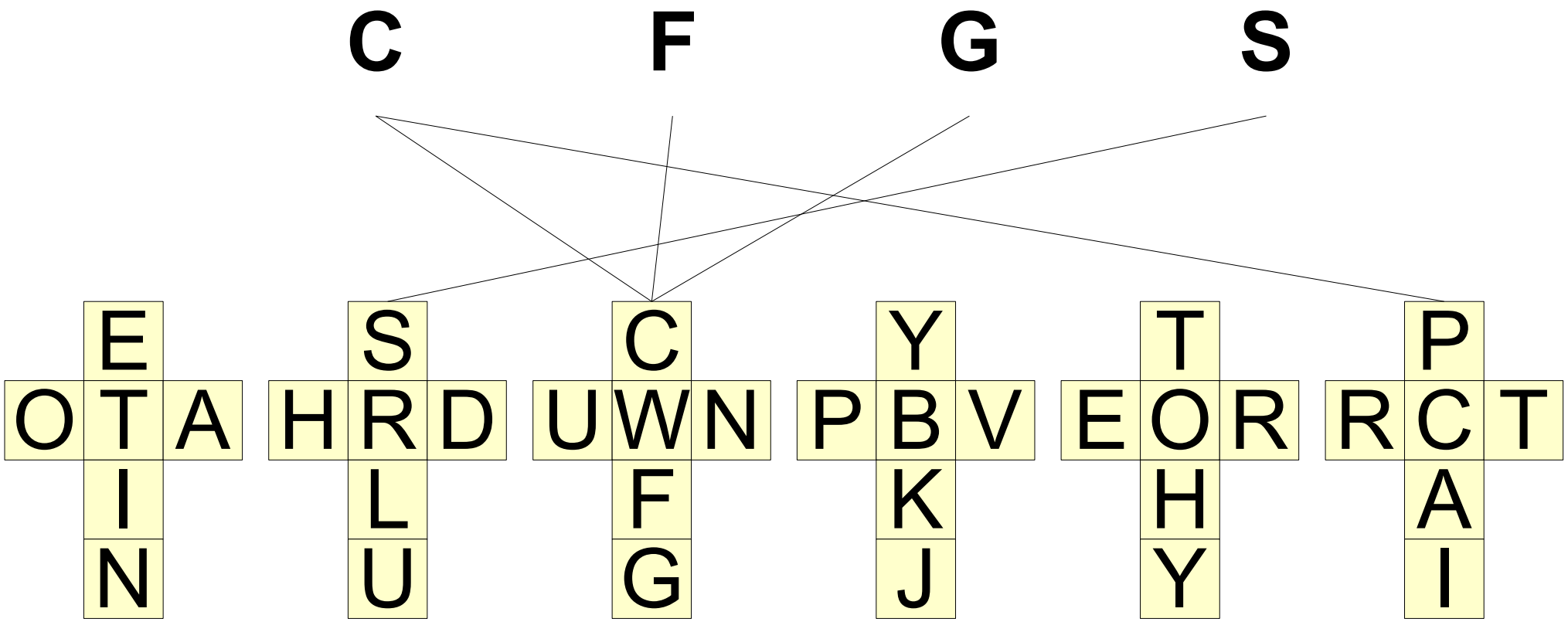
| | | |
|---|---|---|
| | C | |
| U | W | N |
| | F | |
| | G | |

| | | |
|---|---|---|
| | Y | |
| P | B | V |
| | K | |
| | J | |

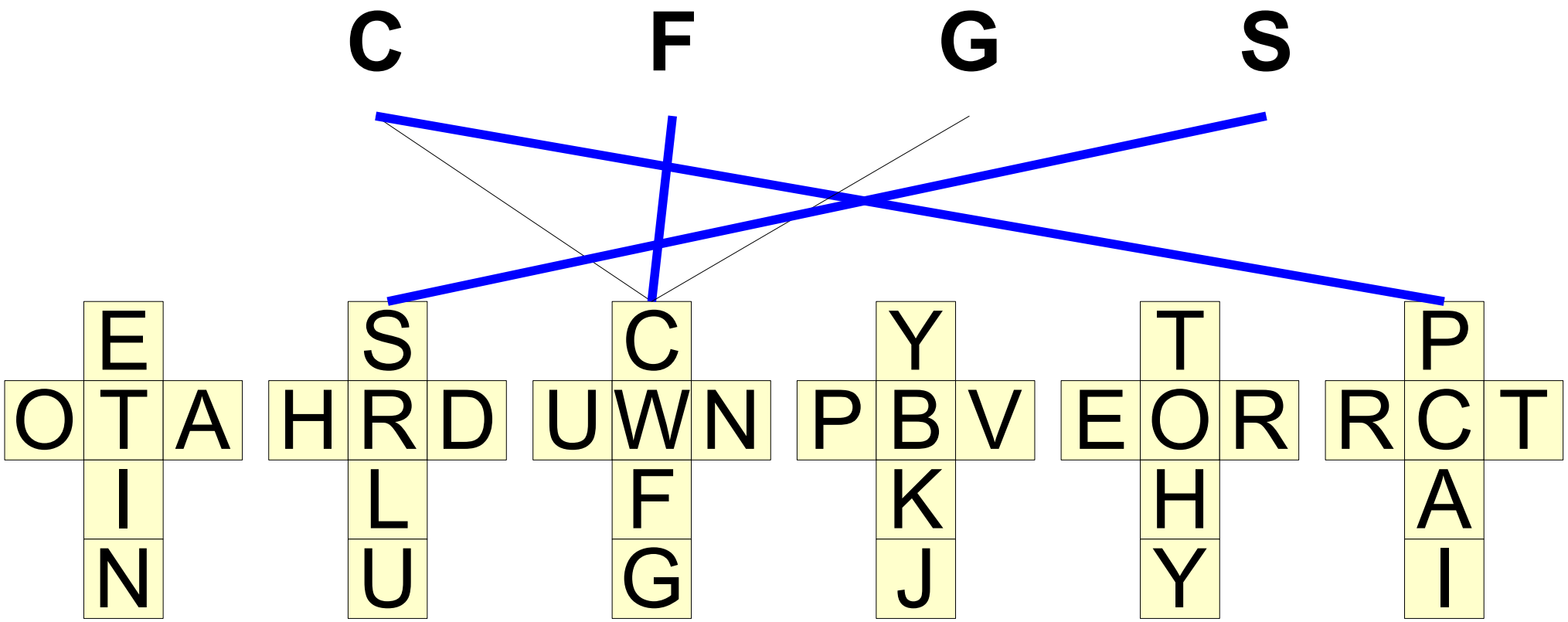
| | | |
|---|---|---|
| | T | |
| E | O | R |
| | H | |
| | Y | |

| | | |
|---|---|---|
| | P | |
| R | C | T |
| | A | |
| | I | |

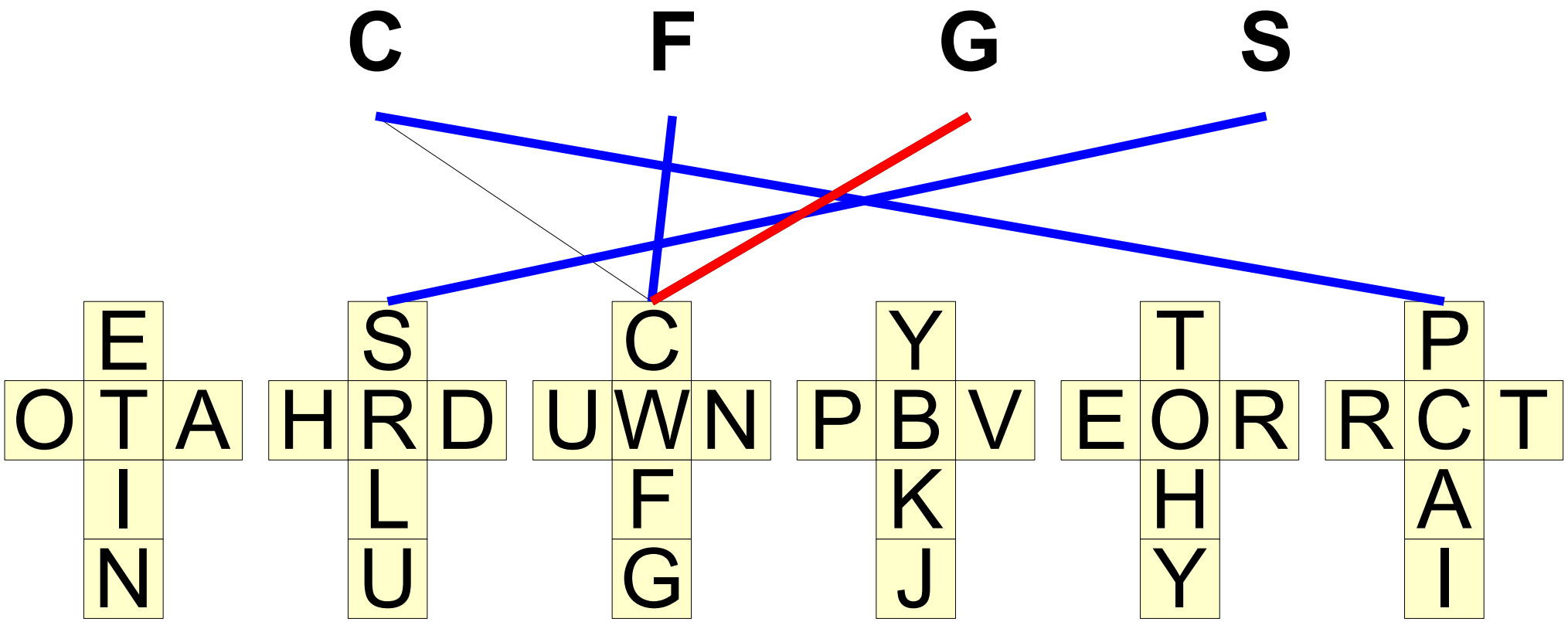
# Boggle Spelling



# Boggle Spelling



# Boggle Spelling



# In Pseudocode

```
boolean canSpell(List<Cube> cubes,
                 String word) {
   return hasMatching(makeGraph(cubes, word),
                      word.length());
}
```