P and **NP**

Recap from Last Time

The Limits of Decidability

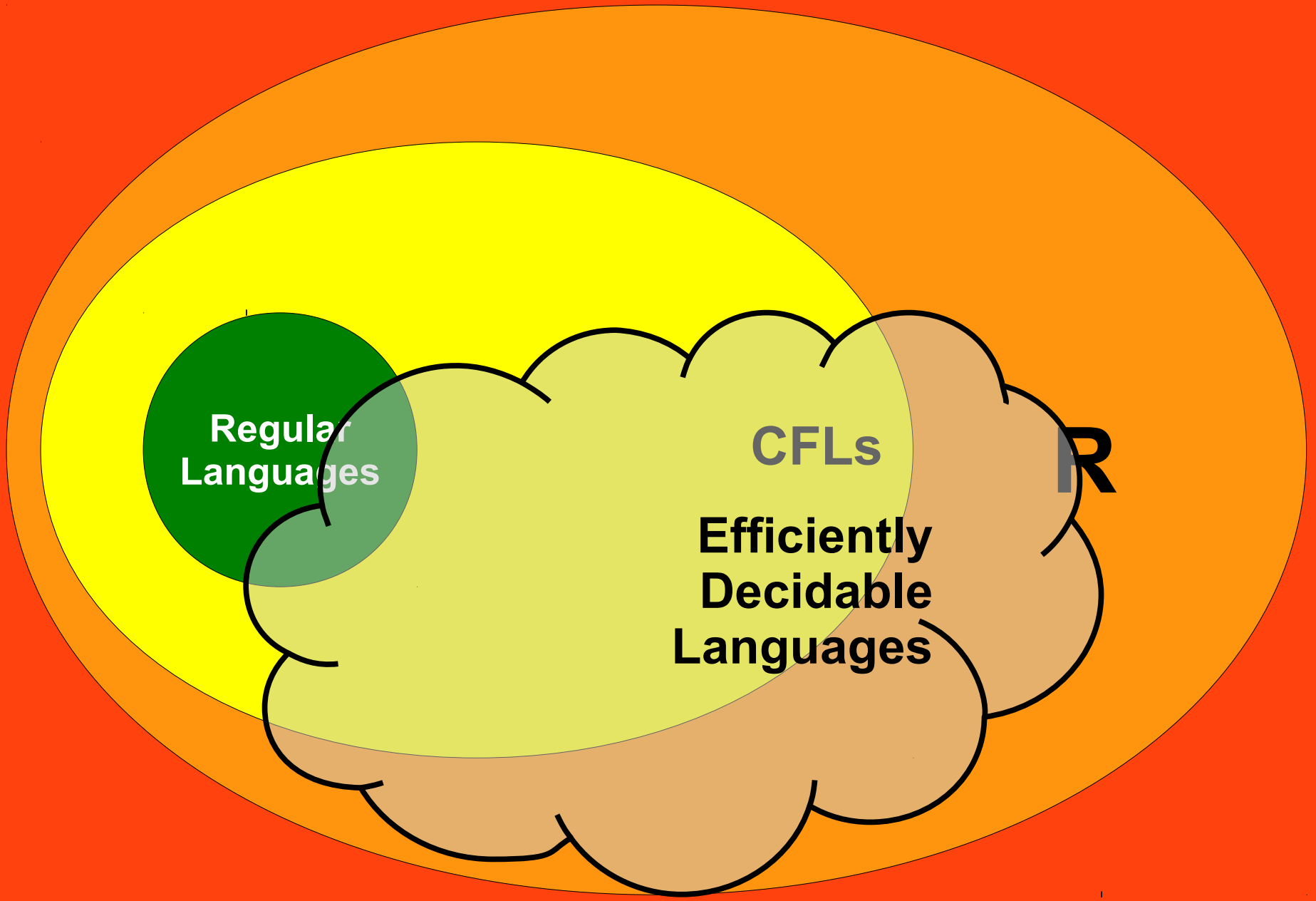
- In ***computability theory***, we ask the question

  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.


Regular Languages
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

What is an efficient algorithm?

For Comparison

- **Longest increasing subsequence:**
  - Naive: $O(n \cdot 2^n)$
  - Fast: $O(n^2)$
- **Shortest path problem:**
  - Naive: $O(n^2 \cdot n!)$
  - Fast: $O(n + m)$, where n is the number of nodes and m the number of edges. (Take CS161 for details!)

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 iff 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

The Complexity Class **P**

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

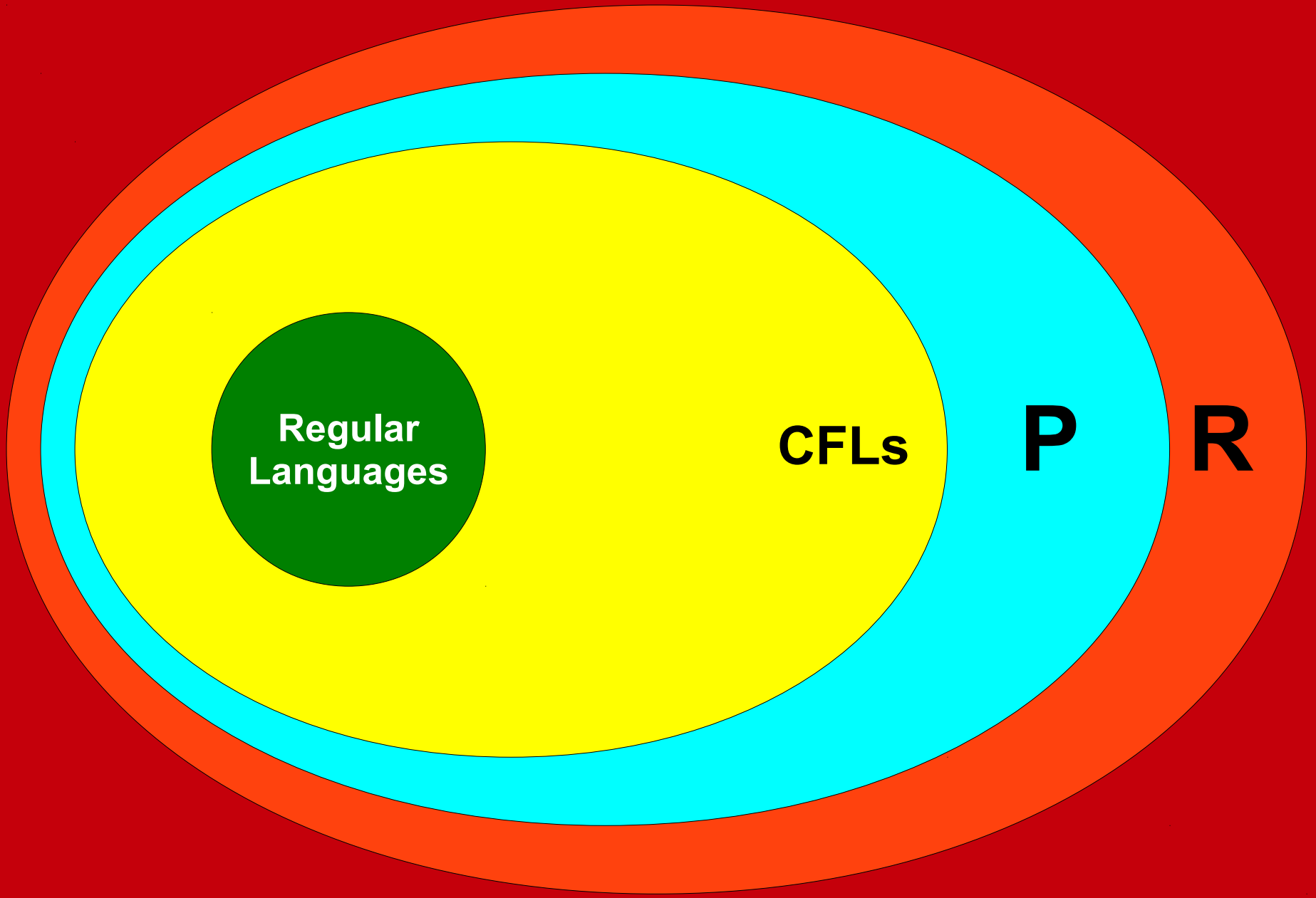
$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$

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

New Stuff!

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.


Regular Languages
CFLs
P
R
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?

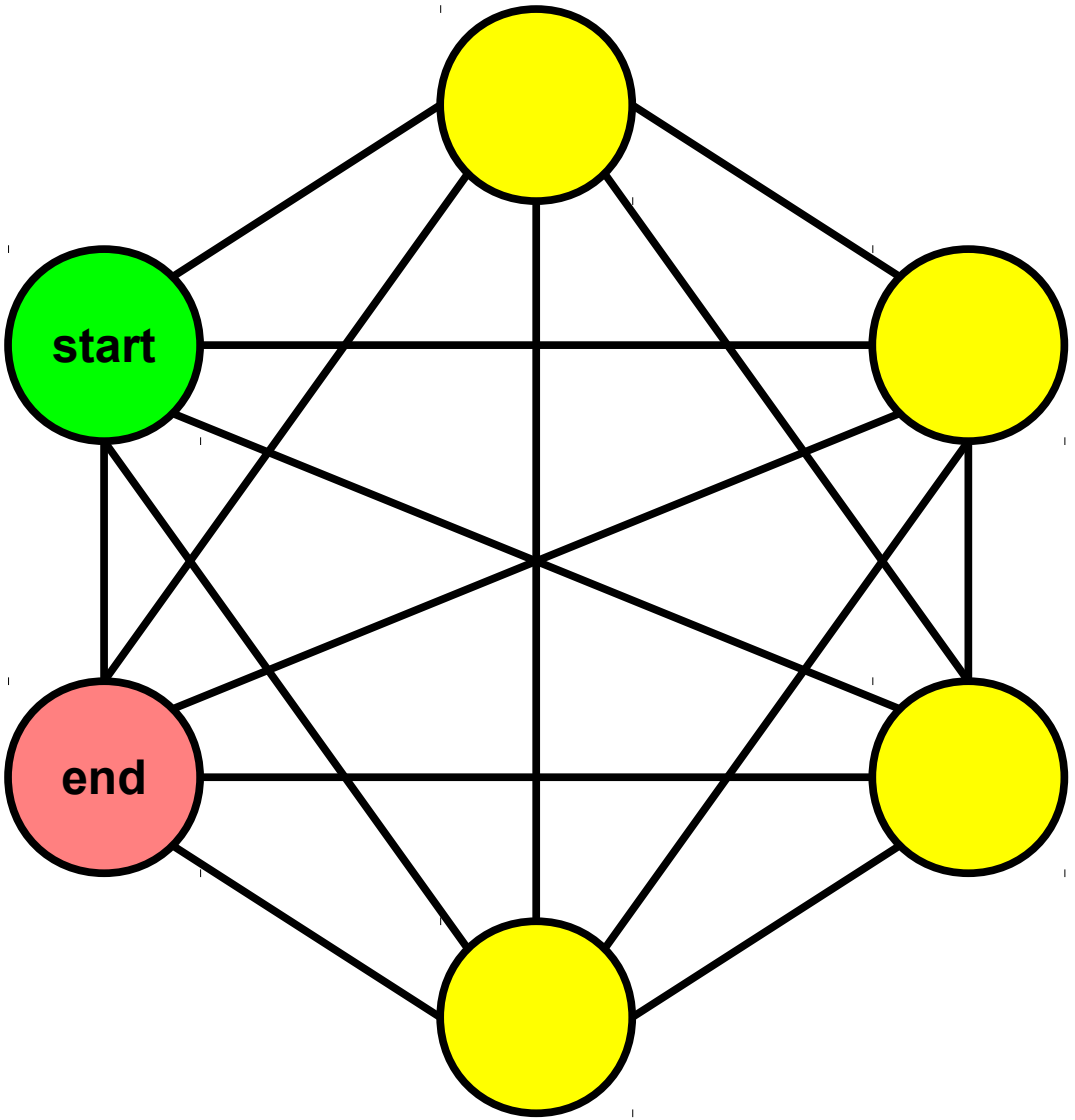
- **Linear programming:**

  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?

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...

What if you could magically guess which element of the search space was the one you wanted?

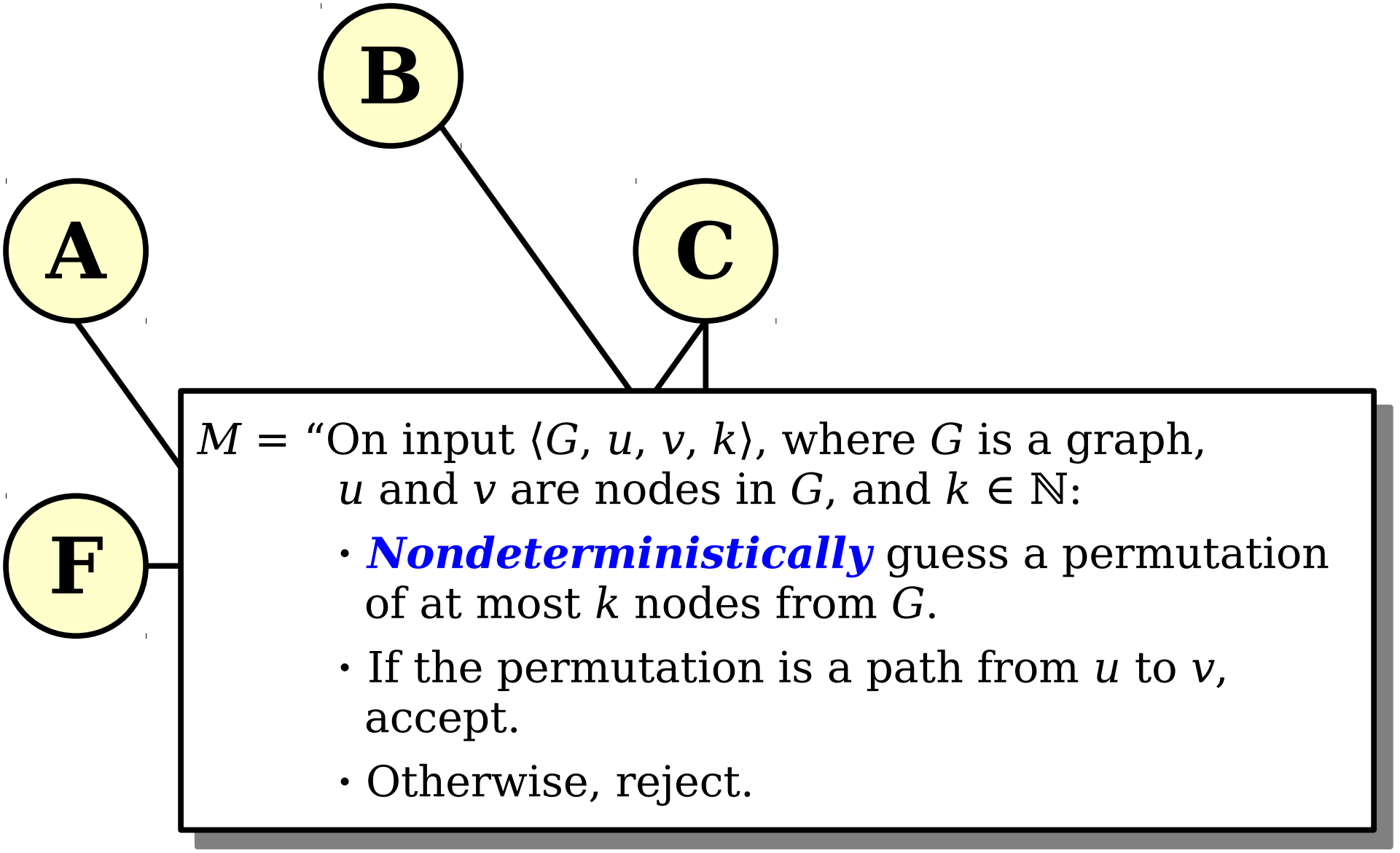
A Sample Problem

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

M = "On input $\langle S, k \rangle$, where S is a sequence of numbers and k is a natural number:

- ***Nondeterministically*** guess a subsequence of S.
- If it is an ascending subsequence of length at least k, accept.
- Otherwise, reject."

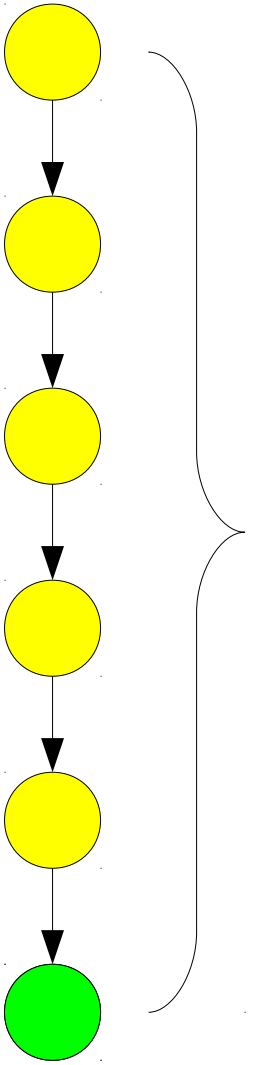
Another Problem



How do we measure NTM efficiency?

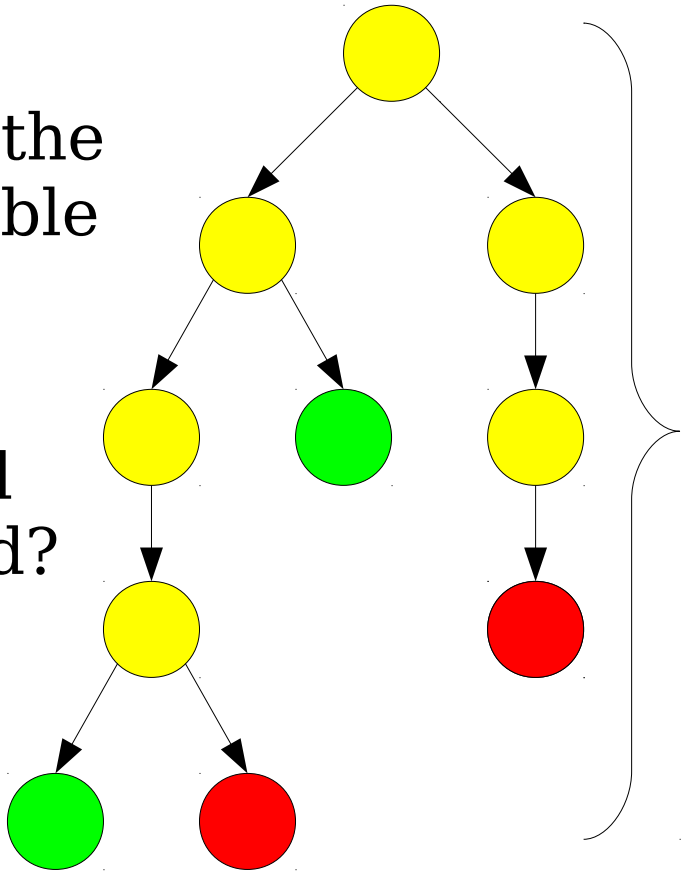
Analyzing NTMs

- When discussing deterministic TMs, the notion of time complexity is (reasonably) straightforward.
- **Recall:** One way of thinking about nondeterminism is as a tree.
- In a ***deterministic*** computation, the tree is a straight line.
- The time complexity is the height of that straight line.

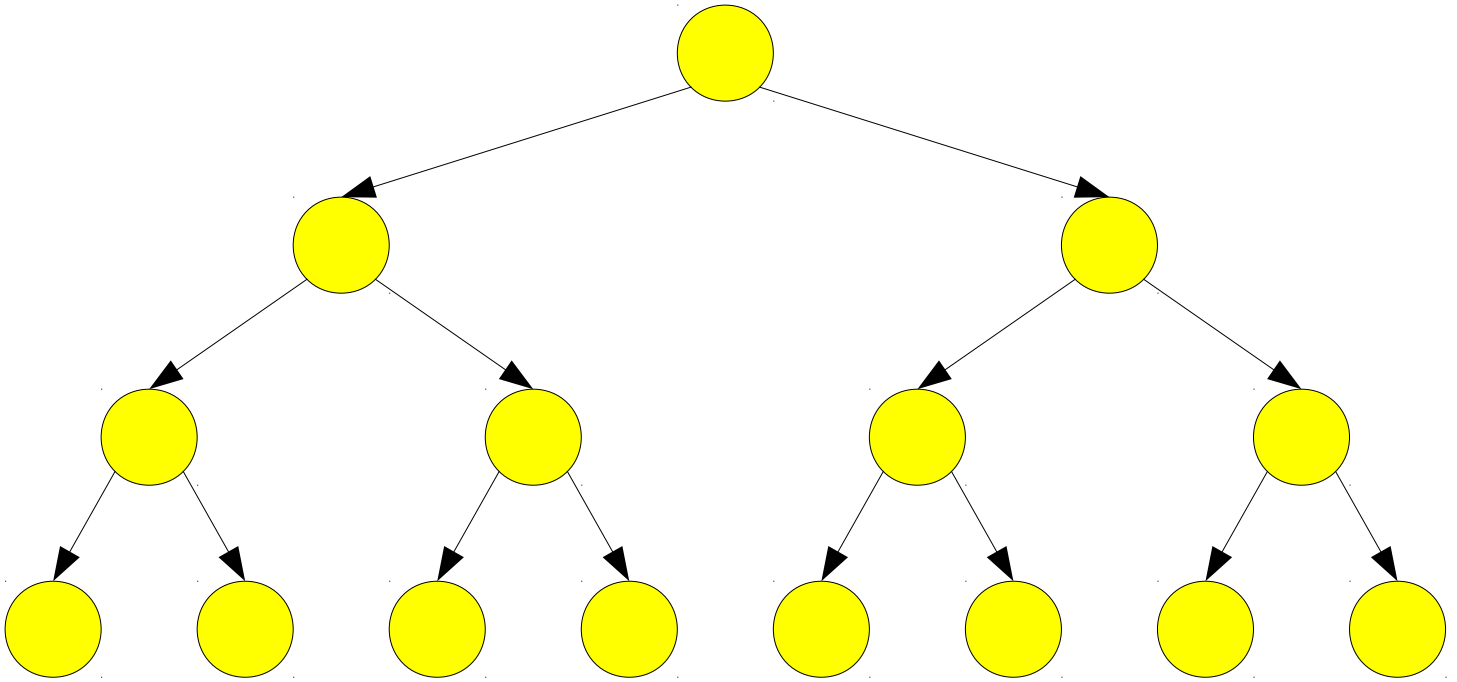


Analyzing NTMs

- When discussing deterministic TMs, the notion of time complexity is (reasonably) straightforward.
- **Recall:** One way of thinking about nondeterminism is as a tree.
- The time complexity is the height of the tree (the length of the ***longest*** possible choice we could make).
- Intuition: If you ran all possible branches in parallel, how long would it take before all branches completed?



The Size of the Tree



From NTMs to TMs

- ***Theorem**:* For any NTM with time complexity $f(n)$, there is a TM with time complexity $2^{O(f(n))}$.
- ***It is unknown whether it is possible to do any better than this in the general case.***
- NTMs are capable of exploring multiple options in parallel; this "seems" inherently faster than deterministic computation.

The Complexity Class **NP**

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

  NP = { L | There is an NTM that decides L in non-deterministic polynomial time. }

- What types of problems are in **NP**?

A Problem in **NP**

- Does an $n^2 \times n^2$ Sudoku grid have a solution?
  - M = "On input $\langle S \rangle$, an encoding of a Sudoku puzzle:
    - ***Nondeterministically*** guess how to fill in all the squares.
    - ***Deterministically*** check whether the guess is correct.
    - If so, accept; if not, reject."

For an arbitrary $n^2 \times n^2$ grid:

Total number of cells in the grid: $\boldsymbol{n^4}$

Total time to fill in the grid: $\boldsymbol{O(n^4)}$

Total number of rows, columns, and boxes to check: $\boldsymbol{O(n^2)}$

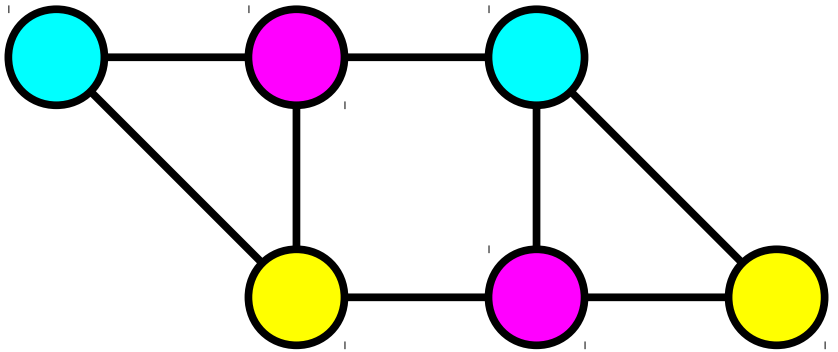
Total time required to check each row/column/box: $\boldsymbol{O(n^2)}$

Total runtime: $\boldsymbol{O(n^4)}$

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

A Problem in **NP**

- A ***k-coloring*** of an undirected graph G is a way of assigning one if k colors to each node in G such that no two nodes joined by an edge have the same color.
  - Applications in compilers, cell phone towers, etc.
- **Question:** Given a graph G and a number k, is graph G k-colorable?
- M = "On input $\langle G, k \rangle$:
  - ***Nondeterministically*** guess a k-coloring of the nodes of G.
  - ***Deterministically*** check whether it is legal.
  - If so, accept; if not, reject."



Other Problems in **NP**

- **Subset sum:**

  Given a set S of natural numbers and a target number n, is there a subset of S that sums to n?

- **Longest path:**

  Given a graph G, a pair of nodes u and v, and a number k, is there a simple path from u to v of length at least k?

- **Job scheduling:**

  Given a set of jobs J, a number of workers k, and a time limit t, can the k workers, working in parallel complete all jobs in J within time t?

Problems and Languages

- Abstract question: does a Sudoku grid have a solution?
- Formalized as a language:

SUDOKU = { ⟨*S*⟩ | *S* is a solvable Sudoku grid. }

- In other words:

S is solvable iff $\langle S \rangle \in SUDOKU$

Problems and Languages

- Abstract question: can a graph be colored with k colors?
- Formalized as a language:

  COLOR = { $\langle G, k \rangle$ | G is an undirected graph, $k \in \mathbb{N}$, and G is k-colorable. }

- In other words:

  G is k-colorable iff $\langle G, k \rangle \in COLOR$

An Intuition for **NP**

- Intuitively, a language L is in **NP** if for every $w \in L$, there is an *efficient* way to prove to someone that $w \in L$.
- Analogous to the verifier intuition for **RE**, except that we need to be able to *efficiently* prove strings are in the language.

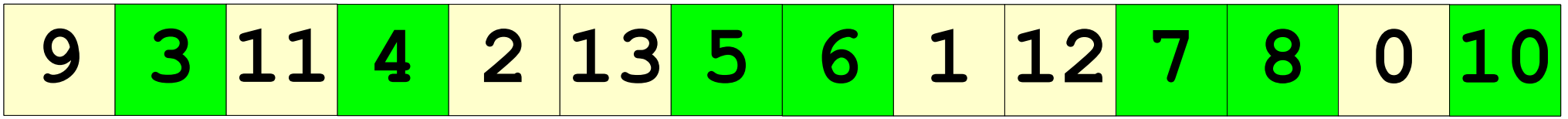
A Problem in **NP**

		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		

A Problem in **NP**

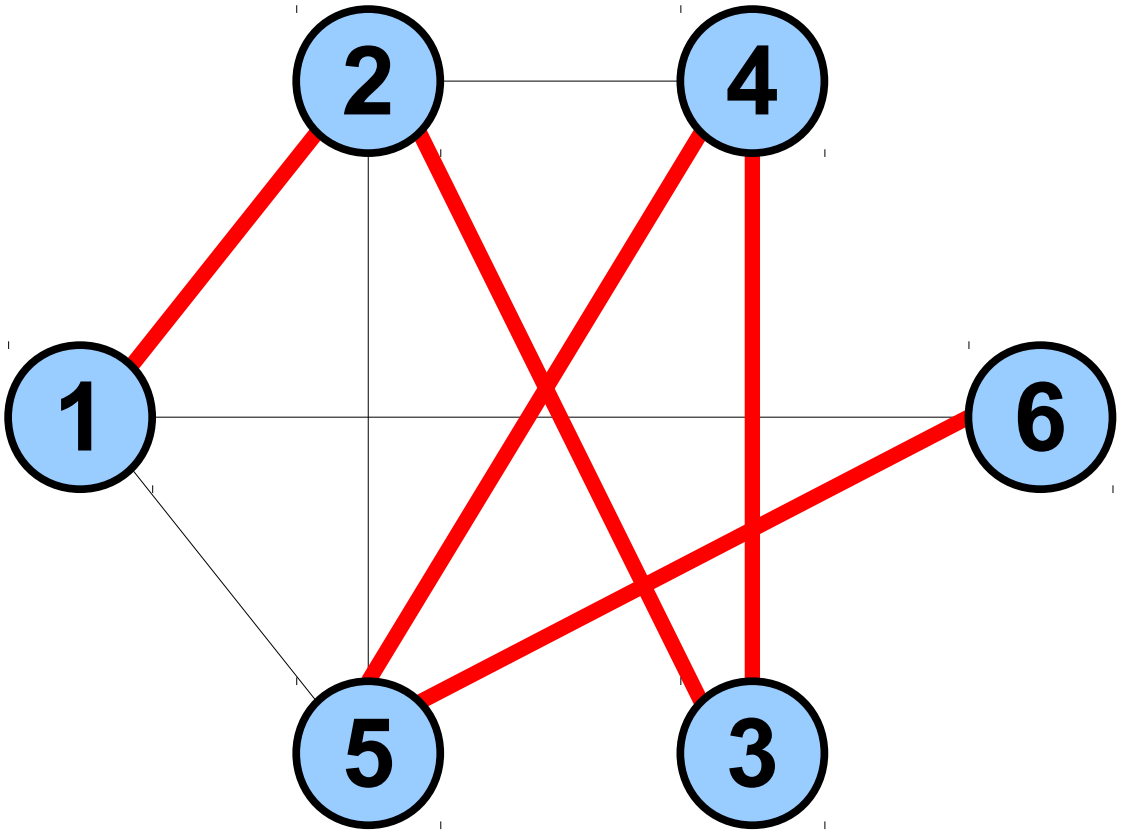
2	5	7	9	6	4	1	8	3
4	9	1	8	7	3	6	5	2
3	8	6	1	2	5	9	4	7
6	4	5	7	3	2	8	1	9
7	1	9	5	4	8	3	2	6
8	3	2	6	1	9	5	7	4
1	6	3	2	5	7	4	9	8
5	7	8	4	9	6	2	3	1
9	2	4	3	8	1	7	6	5

A Problem in **NP**



Is there an ascending subsequence of length at least 7?

A Problem in **NP**



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

Verifiers

- Recall that a ***verifier*** for L is a deterministic TM V such that
  - V halts on all inputs.
  - $w \in L$ iff $\exists c \in \Sigma^*$. V accepts $\langle w, c \rangle$.

- **Theorem:** $L \in \mathbf{RE}$ iff there is a verifier for L.

Polynomial-Time Verifiers

- A ***polynomial-time verifier*** for L is a deterministic 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 $|w|$.
- **Theorem:** $L \in \mathbf{NP}$ iff there is a polynomial-time verifier for L.

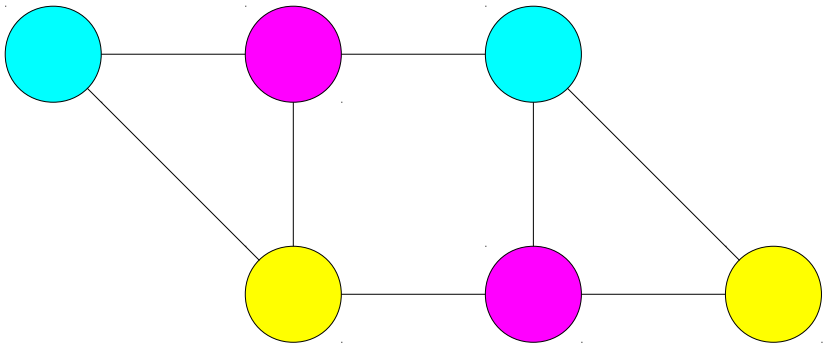
A Problem in **NP**

- Does a Sudoku grid have a solution?
  - M = "On input $\langle S, A \rangle$, an encoding of a Sudoku puzzle and an alleged solution to it:
    - ***Deterministically*** check whether A is a solution to S.
    - If so, accept; if not, reject."

		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		

A Problem in **NP**

- M = "On input $\langle\langle G, k\rangle, C\rangle$, where C is an alleged coloring:
  - ***Deterministically*** check whether C is a legal k-coloring of G.
  - If so, accept; if not, reject."



The Verifier Definition of **NP**

- ***Theorem:*** If there is a polynomial-time verifier V for L, then $L \in \mathbf{NP}$.
- ***Proof idea:*** Build an NTM that nondeterministically guesses a certificate, then deterministically runs the verifier to check it. Then, argue that the NTM runs in nondeterministic polynomial time. ■

The Verifier Definition of **NP**

- ***Theorem:*** If $L \in$ **NP**, there is a polynomial-time verifier for it.
- ***Proof sketch:*** Use the general construction that turns an NTM into a verifier, and argue that the overall construction runs in polynomial time. ■

The

Most Important Question

in

Theoretical Computer Science

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

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

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

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

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 **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 **P** $\stackrel{?}{=}$ **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 **P** $\stackrel{?}{=}$ **NP**.
  - A majority of computer scientists believe **P** ≠ **NP**, but this isn't a large majority.
- Interesting read: Interviews with leading thinkers about **P** $\stackrel{?}{=}$ **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**.

Time-Out for Announcements!

Problem Sets

- Problem Set Seven was due at the start of class today.
  - Want to use a late day? Turn it in tomorrow by 12:50PM.
  - Want to use *two* late days? Turn it in on Friday by 12:50PM.
- Problem Set Eight goes out now, is due one week from today at 12:50PM.
  - Explore the limits of **RE** languages, the **P** vs. **NP** question, and the Big Picture.
  - ***No late days may be used on this assignment***. It's university policy; sorry about that!

Your Questions

“How much math do I need for the different areas of CS? Should I take linear algebra? Partial differential equations? (I've heard they're used in graphics?) Statistics (for ML)? Game theory (also for ML)? Real analysis?”

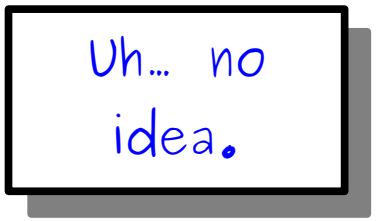
It really depends on the field you're in. Everyone needs linear algebra – it's really useful! Math 51 does a bit of this, but courses like Math 104, EE103, CS205A, and Math 113 would all be good here.

If you're thinking about doing AI, I'd recommend getting a lot of background in linear algebra and statistics. If you want to do graphics, take more linear algebra, geometry, and topology. For crypto, take number theory and abstract algebra. For algorithms, combinatorics is quite useful. Convex optimization is also a really nice tool to have. Game theory and real analysis are probably only important if you specifically want to use them; you probably don't need them for AI.

“Where do you see yourself in 5 years?”

That's a tough one. I'm not sure! Probably keeping up academics in some way, shape, or form. I'd like to keep teaching, but there's a part of me that thinks I might try out a Ph.D if I can stomach missing out on teaching.

“Where do you see yourself in 30 years?”



“With the quarter winding down, it's hard to stay motivated while classes start cramming in material. What are some ways to keep our momentum, for this class and in general?”

For starters, hang in there!

For CS103, we're just about done with new topics, even though it doesn't look like it. You're at a really cool point now where you can look back over everything we've done and start to see a bigger picture at work. All of these topics connect and interrelate in interesting and surprising ways. Think about the overarching questions for the class – do you see how they led us here?

For other courses – assuming the course content is not chosen randomly, there's likely a similar story at work. See if you can thread a narrative through what you're learning. How does everything relate? Why are you learning it? And what comes next? Having a sense of where you're going can make things a lot more interesting.

“Your previous talks on nixtamalization and orangeries were fantastic, and really brightened our days! Week nine isn't always easy, but it would be a little easier if you spoke again about something that fascinates you. Please?”

This is a good question, but I don't think we have time for it right now. Ask me again on Friday!

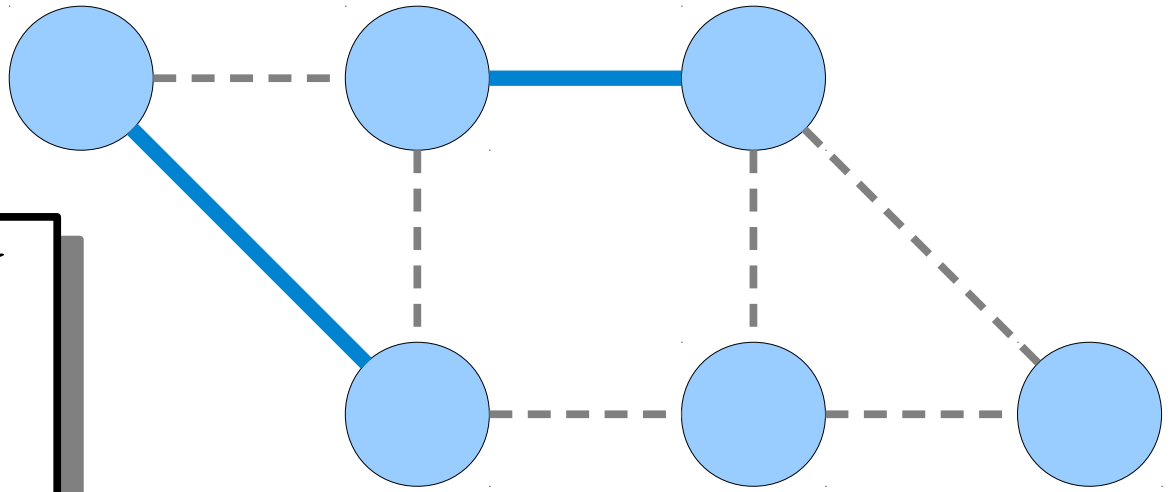
Back to CS103!

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

Reducibility

Maximum Matching

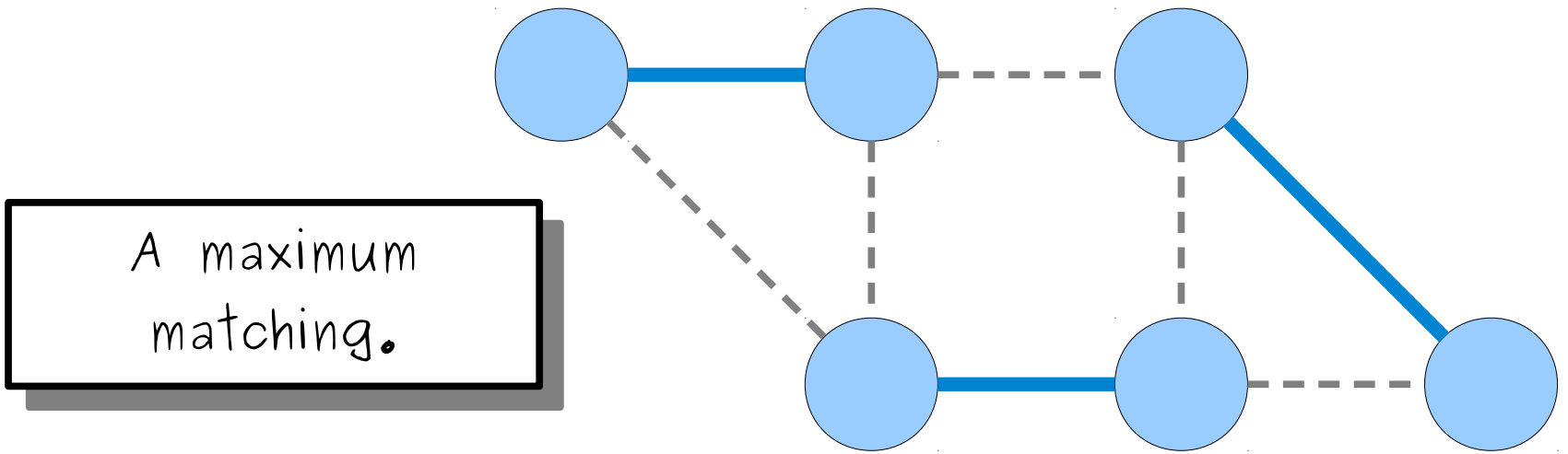
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



A matching, but not a maximum matching.

Maximum Matching

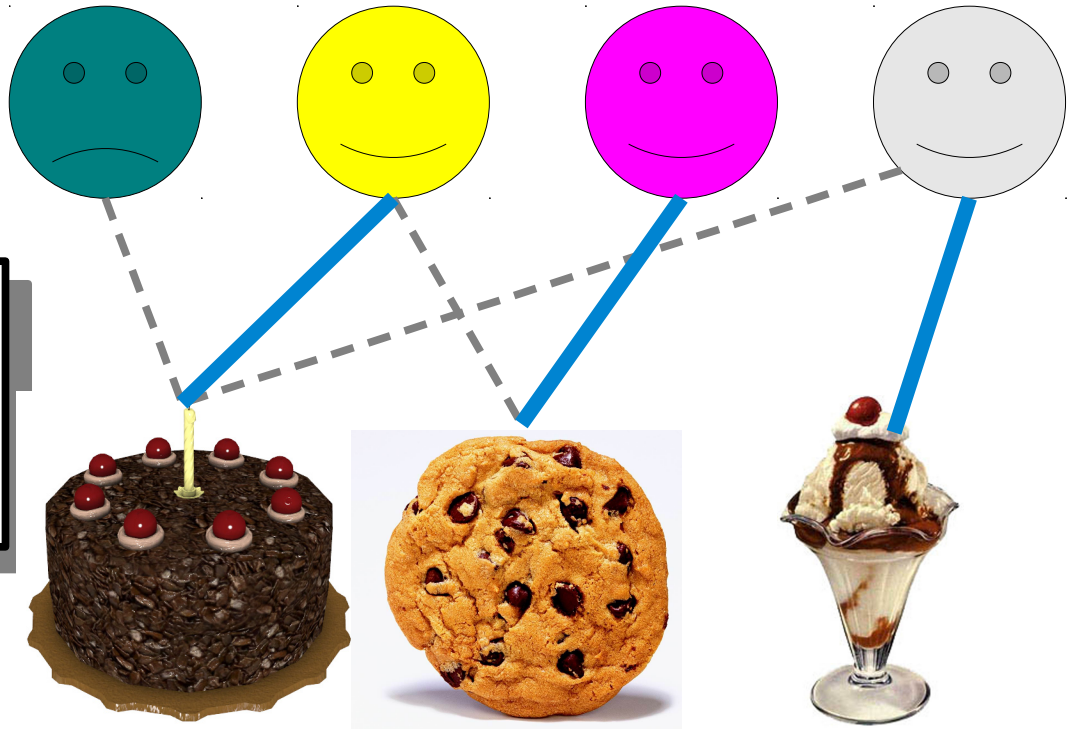
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



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.

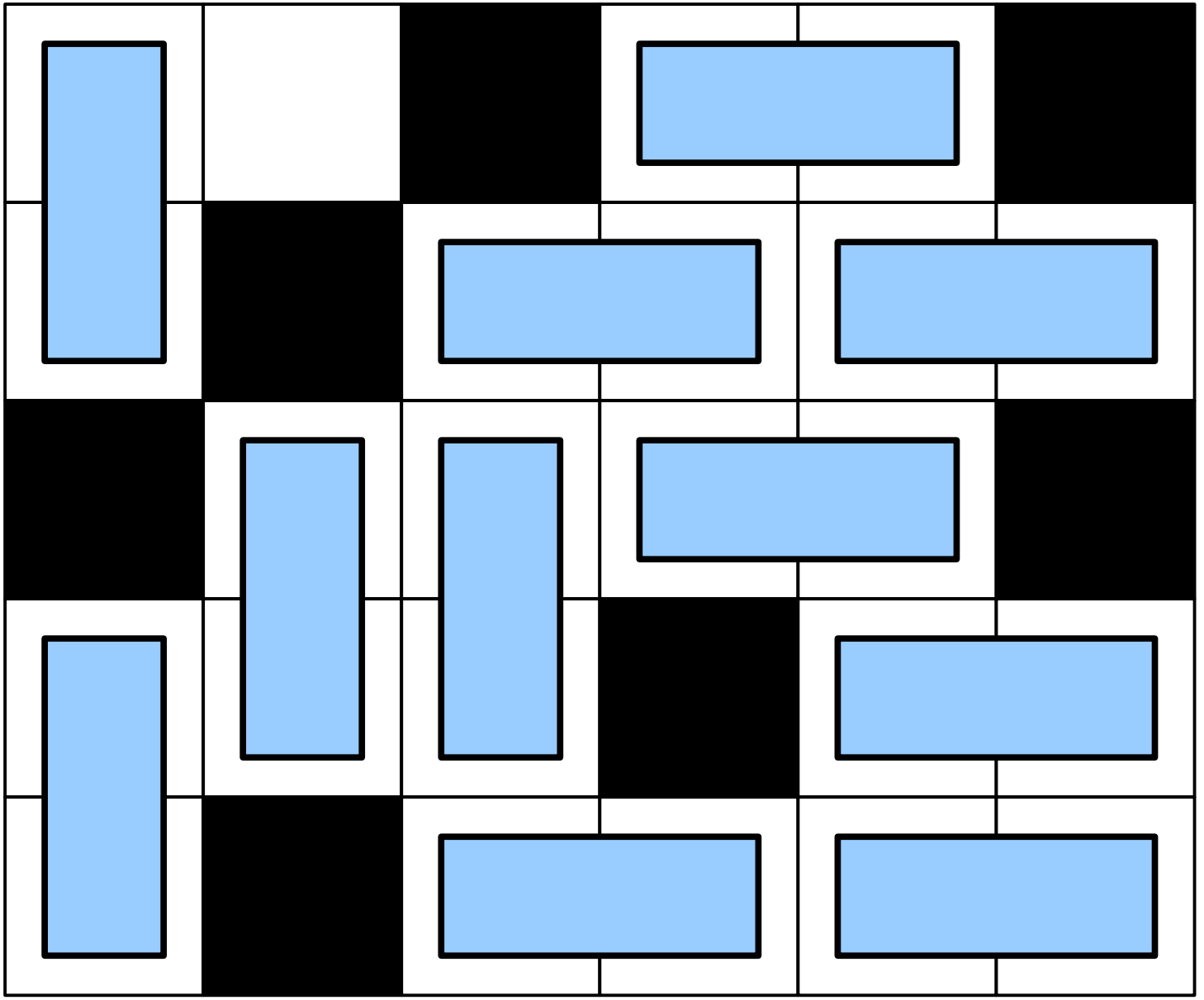
Maximum matchings are not necessarily unique.



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



A Domino Tiling Reduction

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

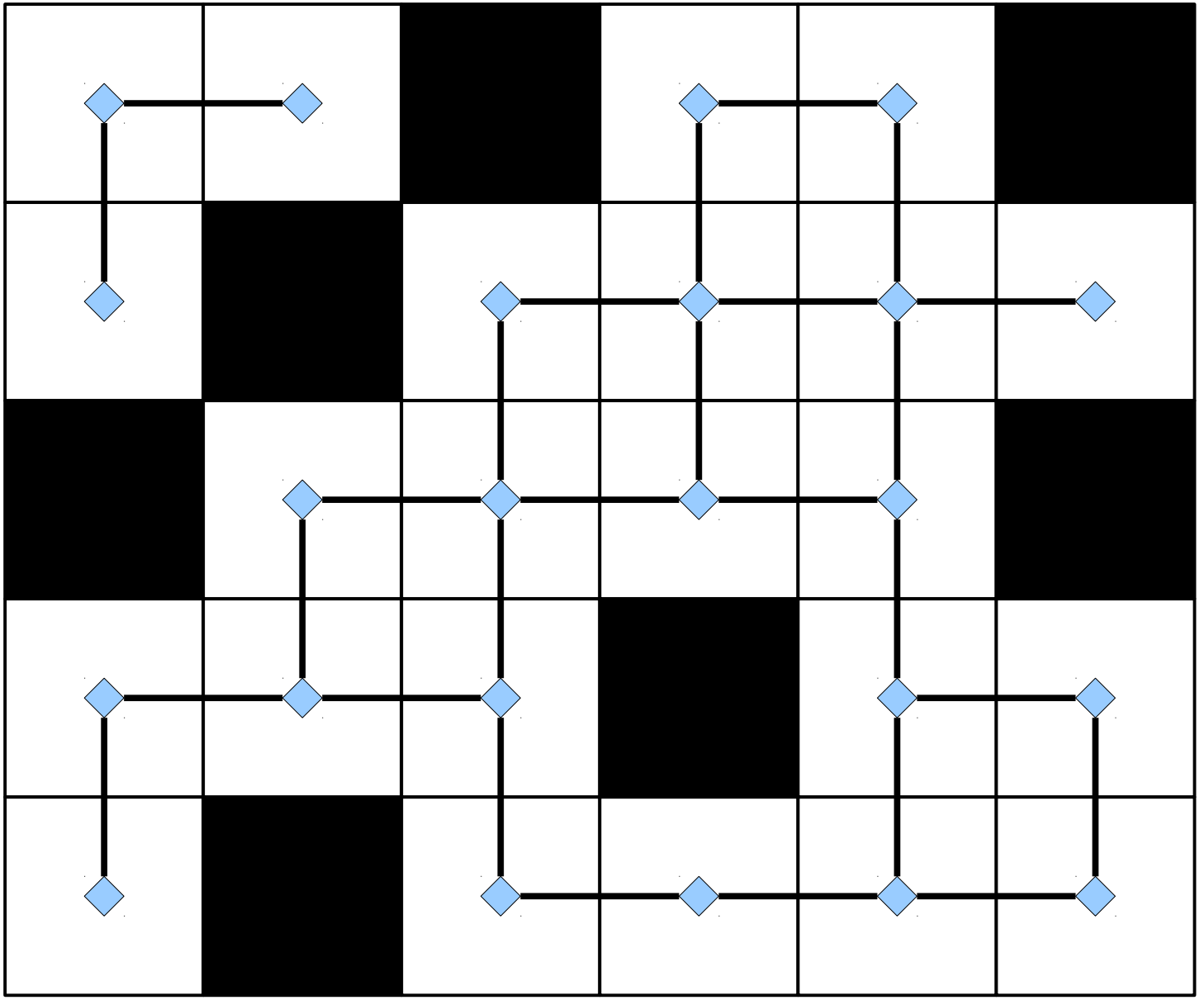
  $MATCHING = \{ \langle G, k \rangle \mid G$ is an undirected graph with a matching of size at least $k \}$

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

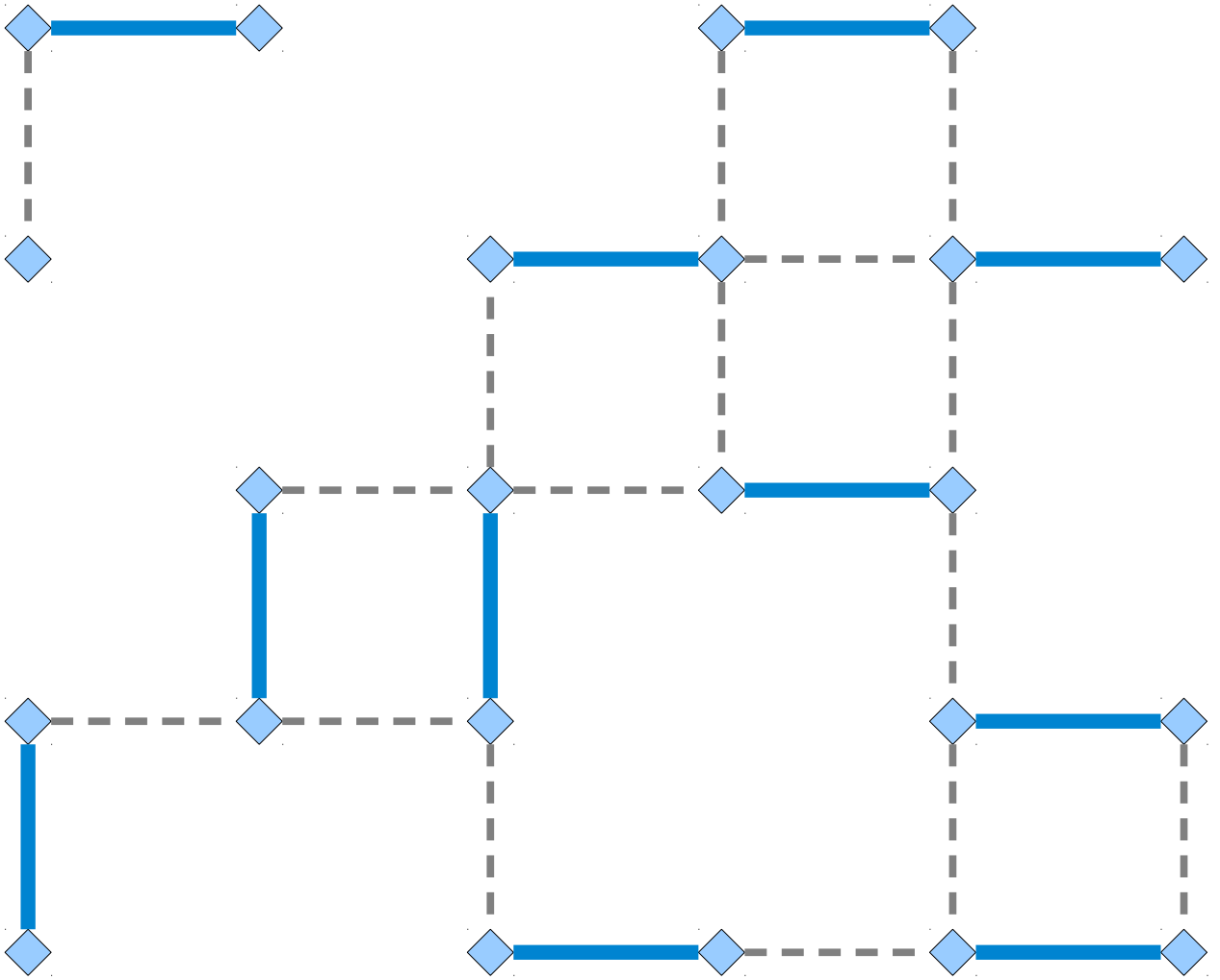
  $DOMINO = \{ \langle D, k \rangle \mid D$ is a grid and k nonoverlapping dominoes can be placed on D. $\}$

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

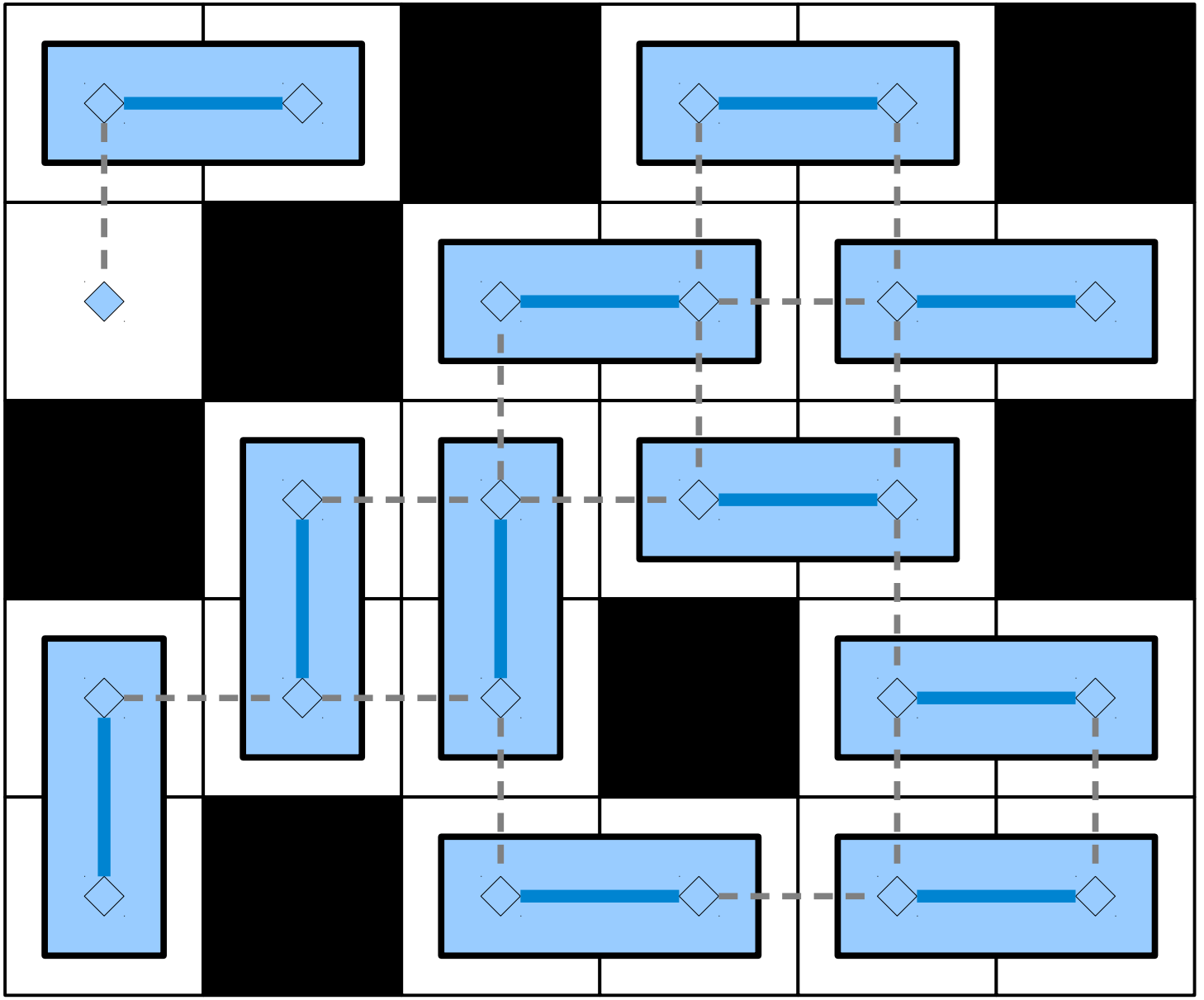
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



The Setup

- To determine whether you can place at least k dominoes on a crossword grid, do the following:
  - Convert the grid into a graph: each empty cell is a node, and any two adjacent empty cells have an edge between them.
  - Ask whether that graph has a matching of size k or greater.
  - Return whatever answer you get.

In Pseudocode

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

Why This Works

- This overall construction gives a polynomial-time algorithm for the domino tiling problem.
- It takes on polynomial time to convert the grid into a graph (we'll hand-wave these details away.)
- Once we have that new graph, it takes only polynomial time to check if there's a sufficiently large matching.
- Overall, this only requires polynomial time.