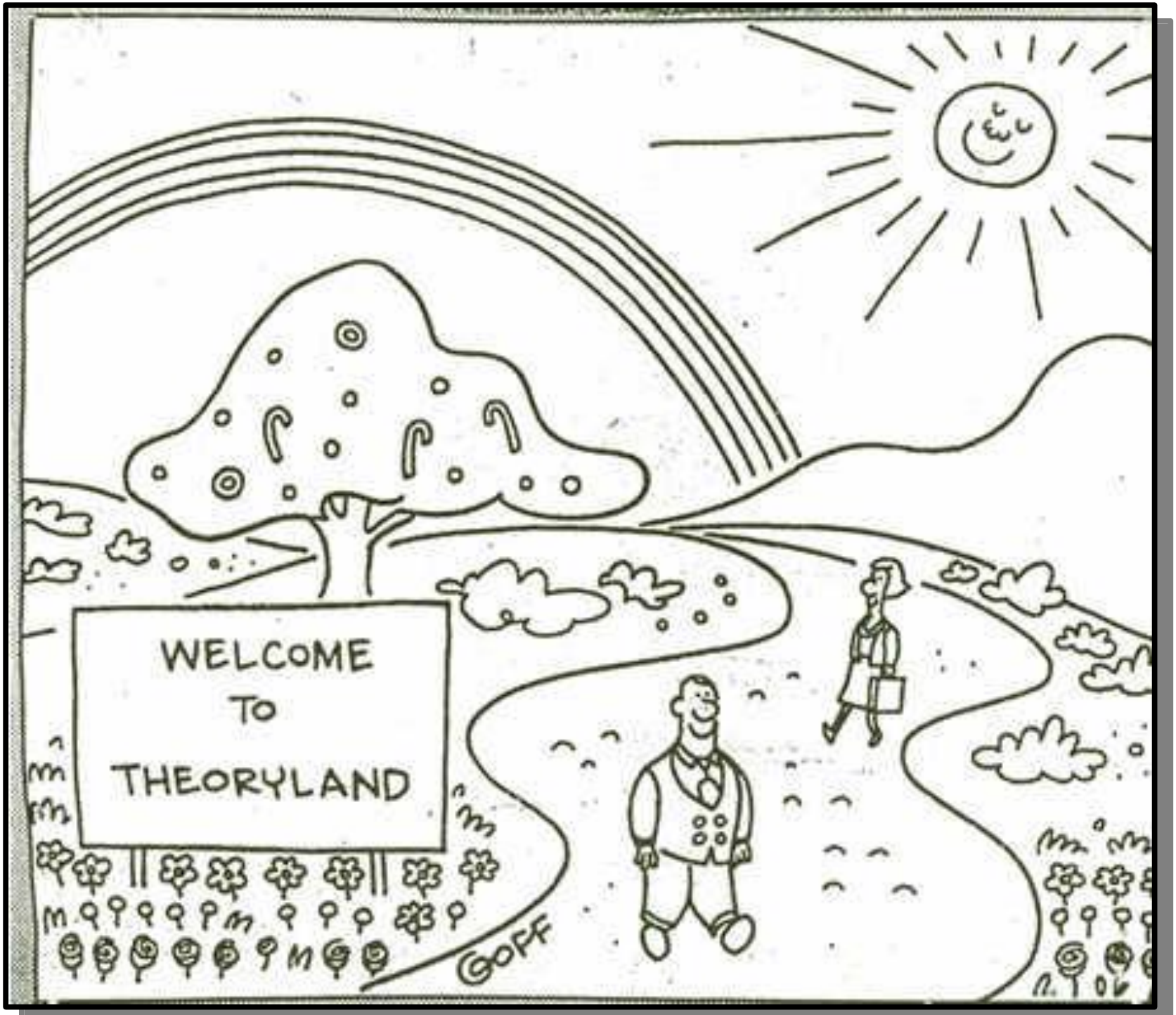
Complexity Theory

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.

$$\forall x.\ x + 1 \neq 0$$

$$\forall x.\ \forall y.\ (x + 1 = y + 1 \rightarrow x = y)$$

$$\forall x.\ x + 0 = x$$

$$\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$$

$$(P(0) \wedge \forall y.\ (P(y) \rightarrow P(y + 1))) \rightarrow \forall x.\ P(x)$$

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 $c = 1$.

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

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?

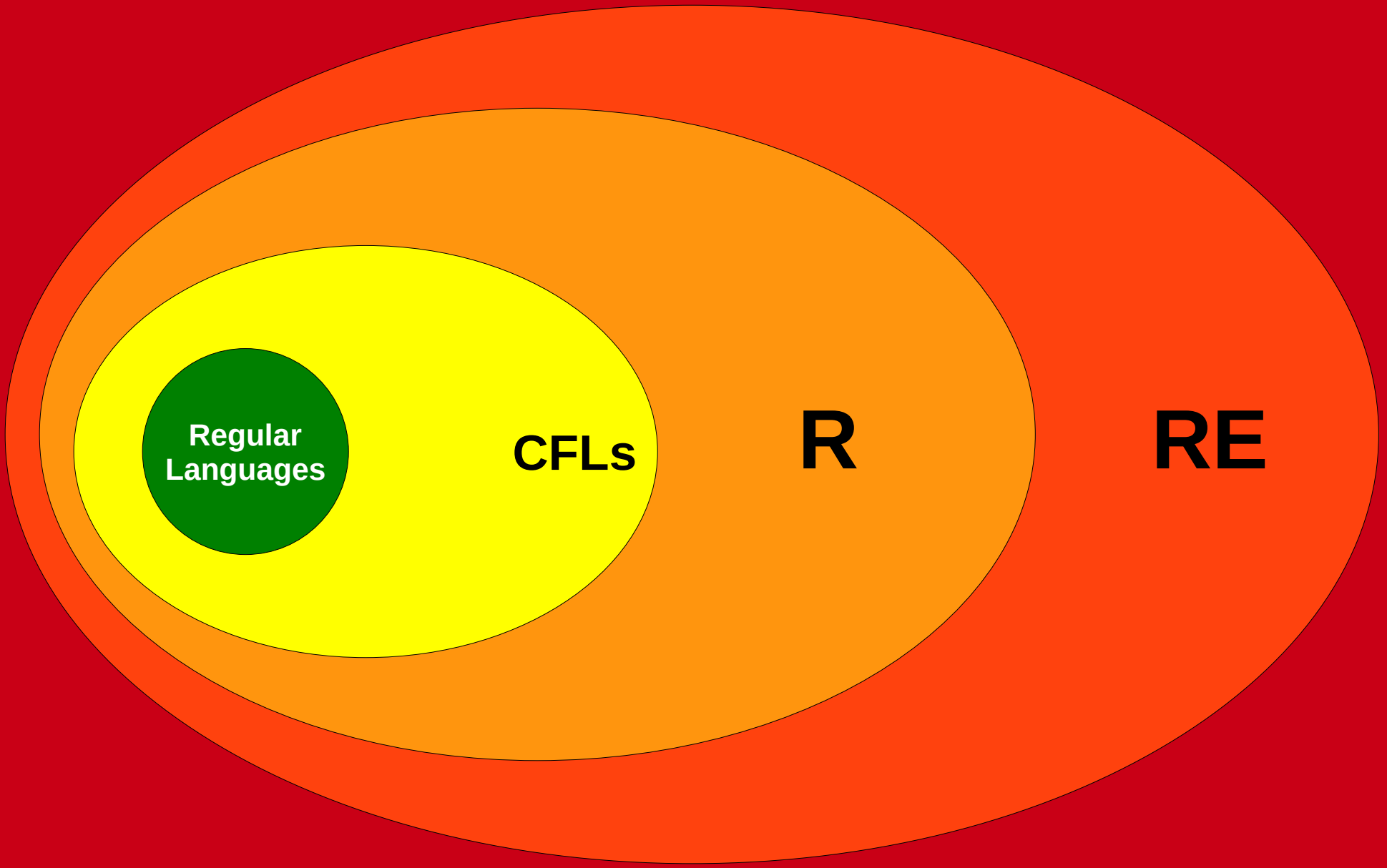
In the remainder of this course, we will explore this question in more detail.

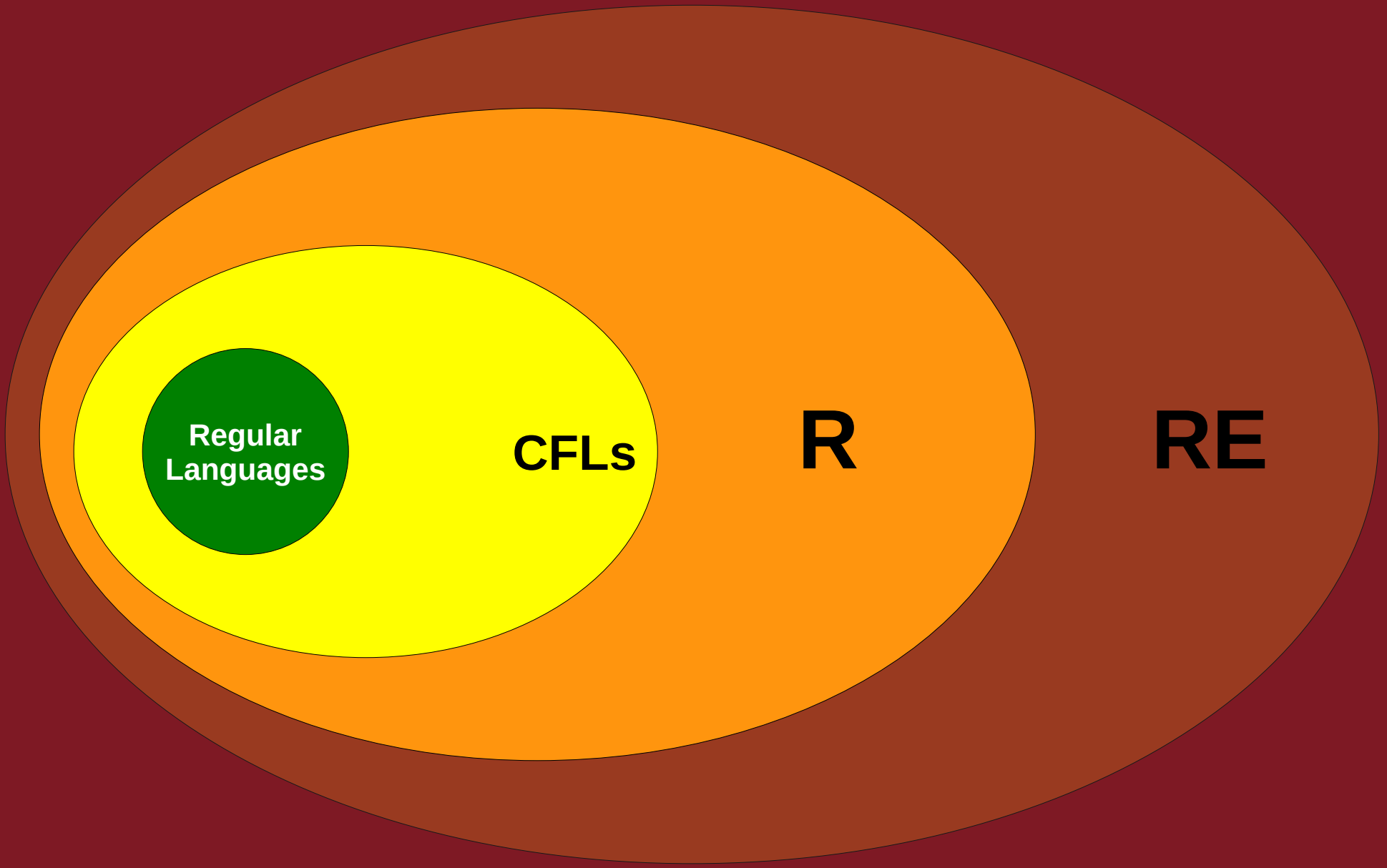
Where We've Been

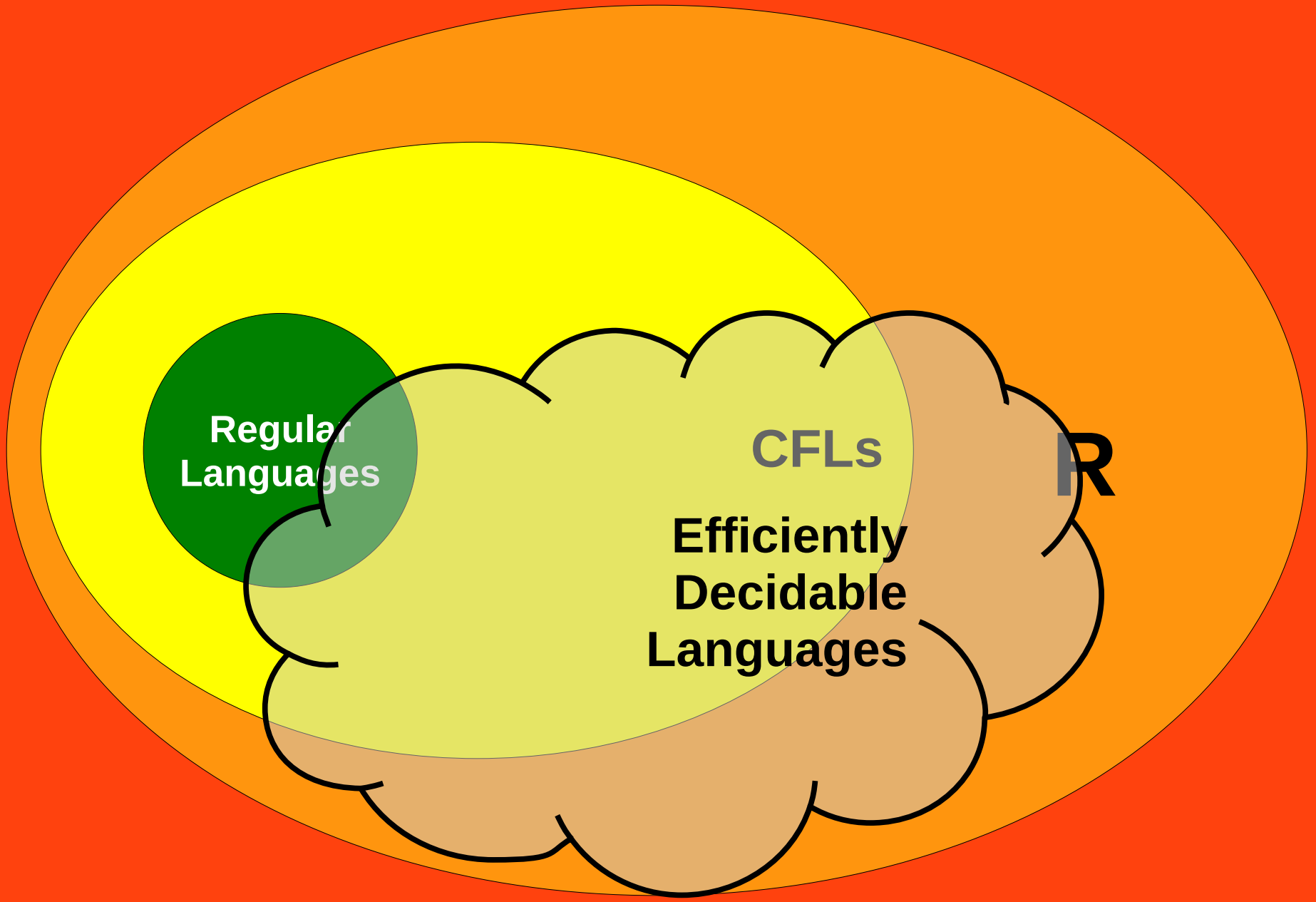
- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.

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
R
RE
All Languages


Regular Languages
CFLs
R
Efficiently Decidable 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?

- Number of states.
- Size of tape alphabet.
- Size of input alphabet.
- Amount of tape required.
- Number of steps 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 may be totally unacceptable.

A Sample Problem

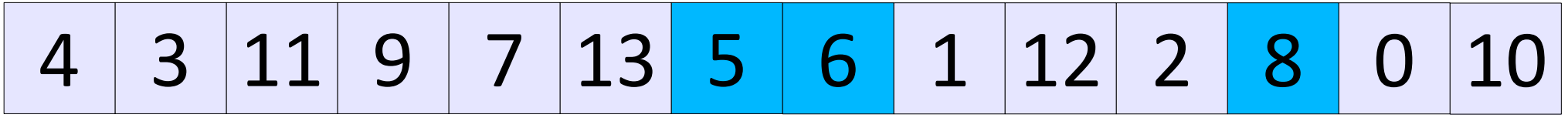
4	3	11	9	7	13	5	6	1	12	2	8	0	10

A Sample Problem

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

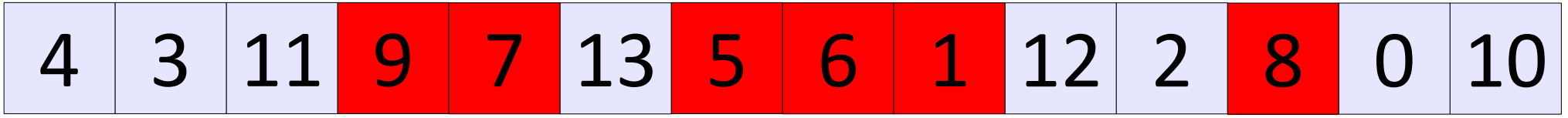
Goal: Find the length of the longest increasing subsequence of this sequence.

A Sample Problem



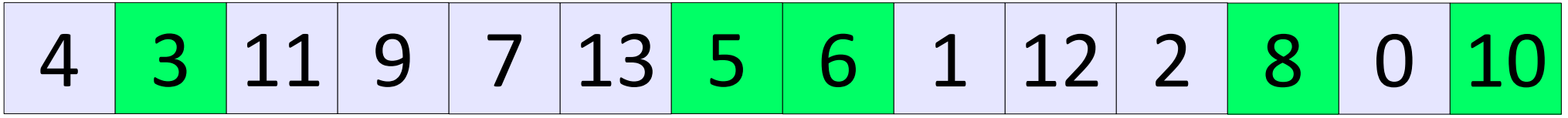
Goal: Find the length of the longest increasing subsequence of this sequence.

A Sample Problem



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

A Sample Problem



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.)

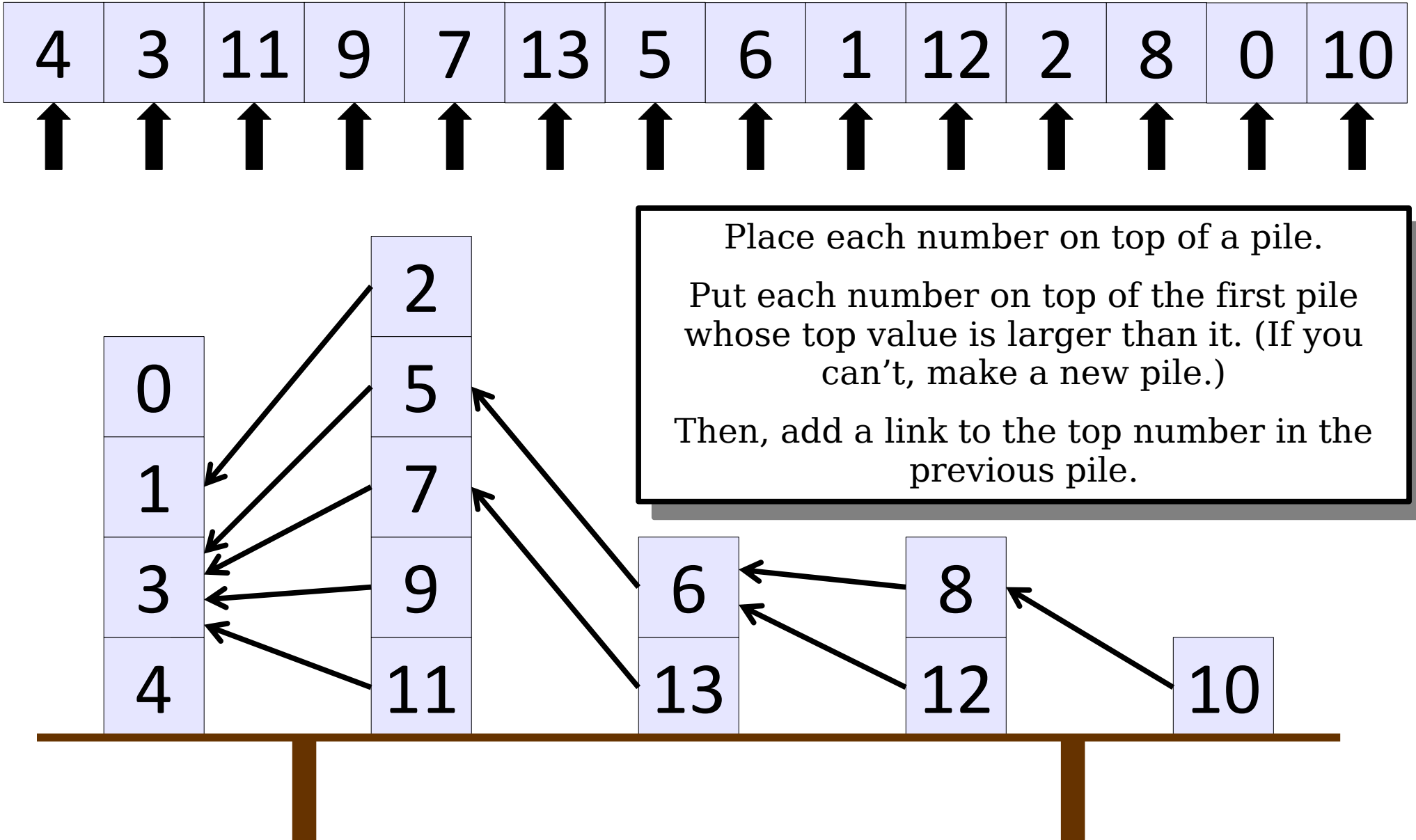
Checking all of them to find the longest increasing subsequence will take time $O(n \cdot 2^n)$.

Nifty fact: the age of the universe is about 4.3×10^{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.

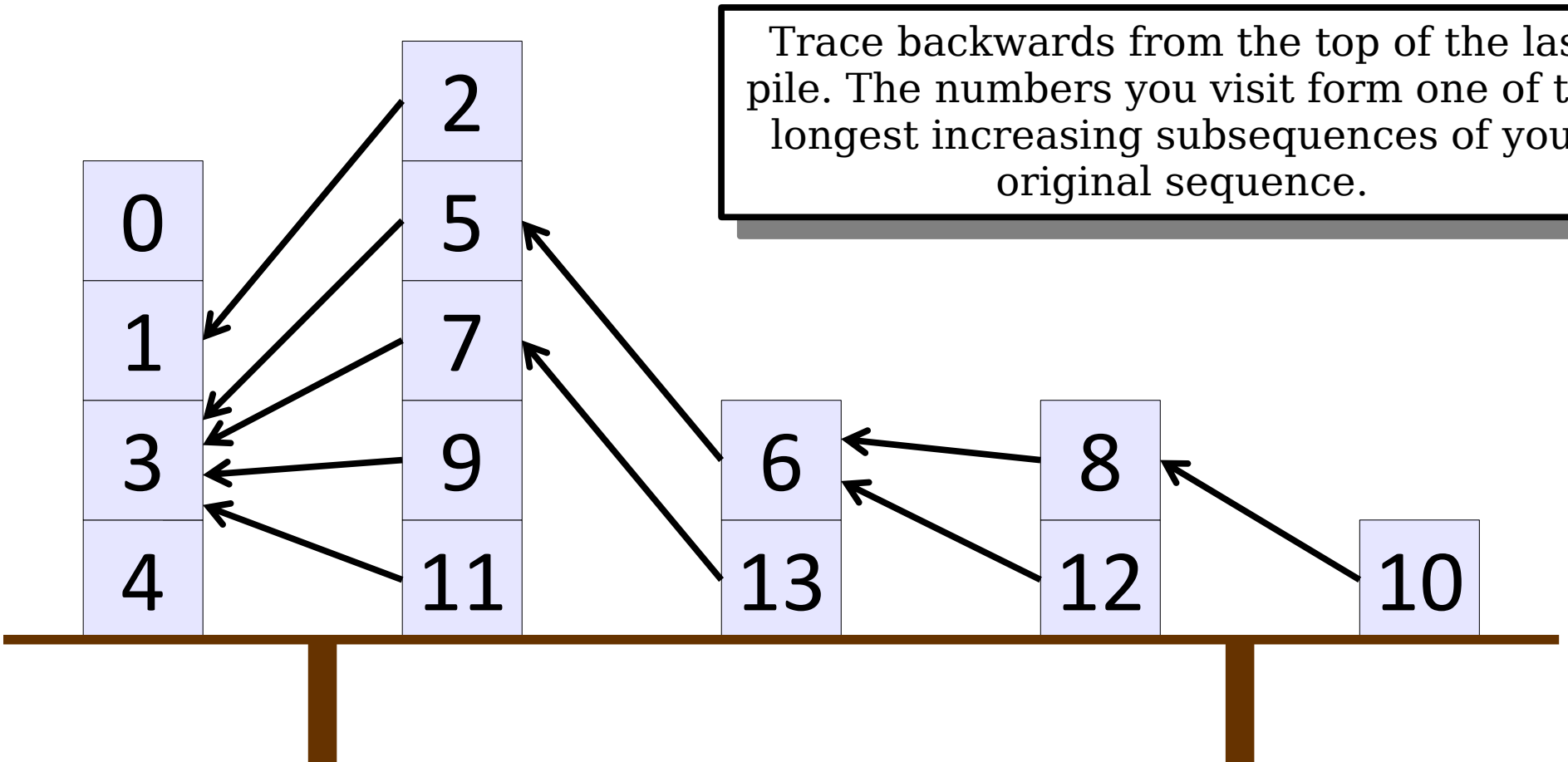
A Different Approach

Patience Sorting



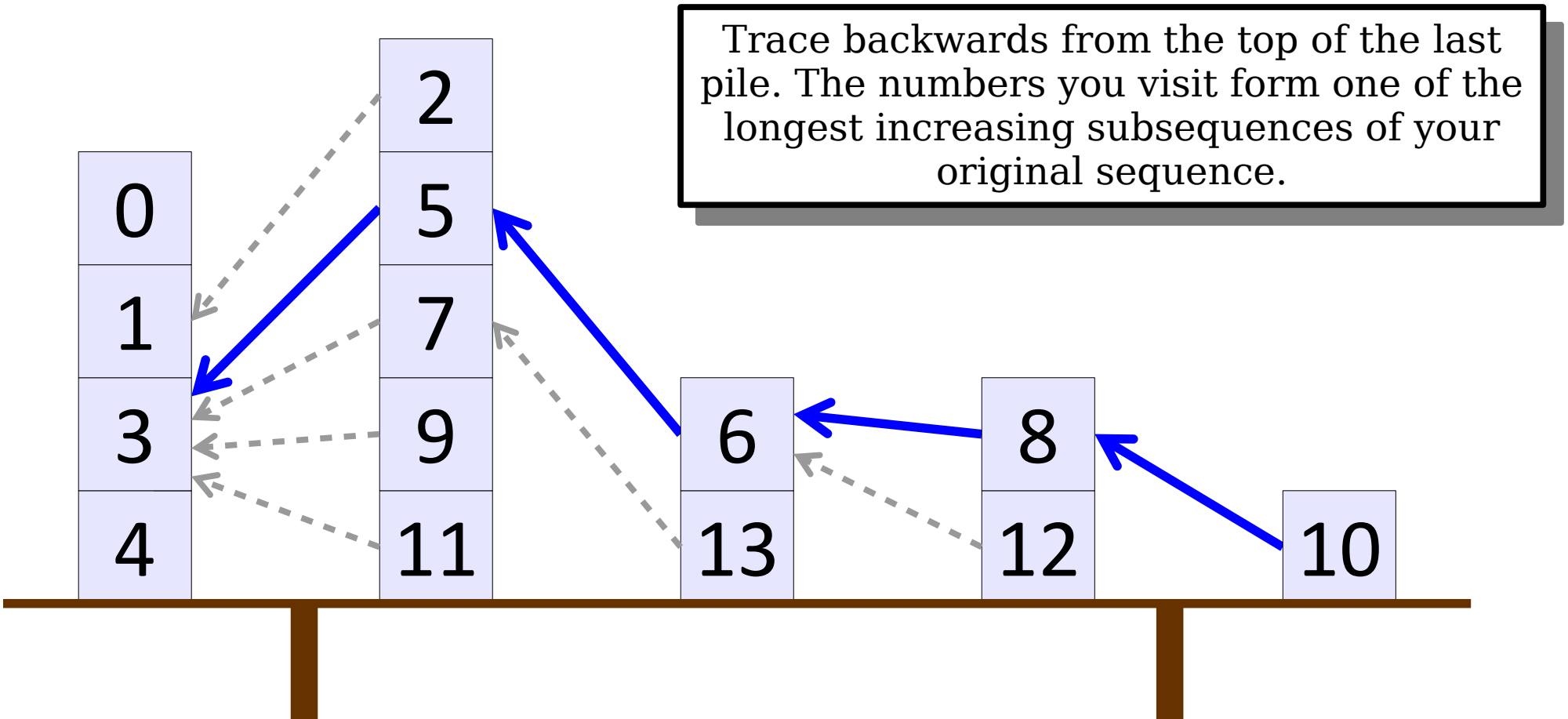
Patience Sorting

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



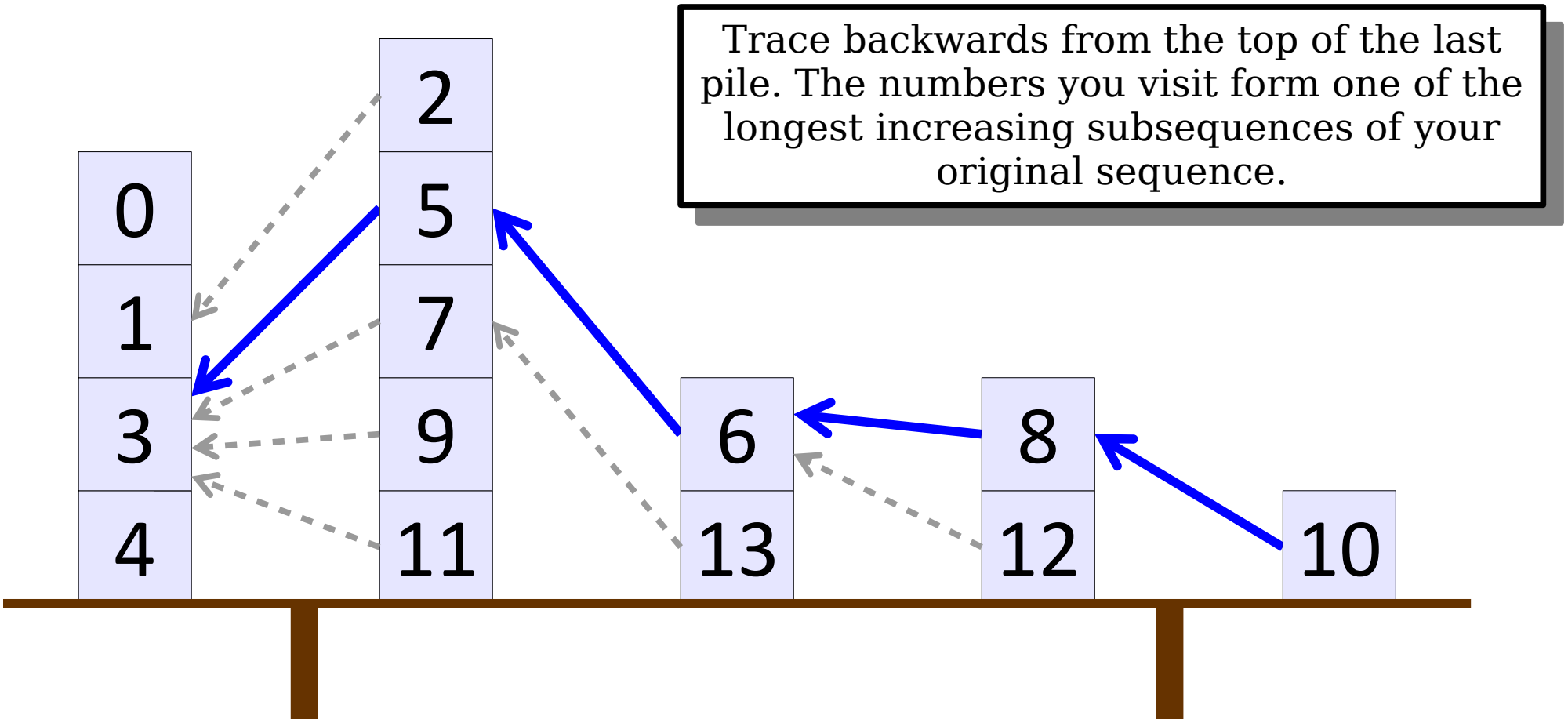
Patience Sorting

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



Patience Sorting

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



Longest Increasing Subsequences

Theorem: There is an algorithm that can find the longest increasing subsequence of an array in time $O(n^2)$.

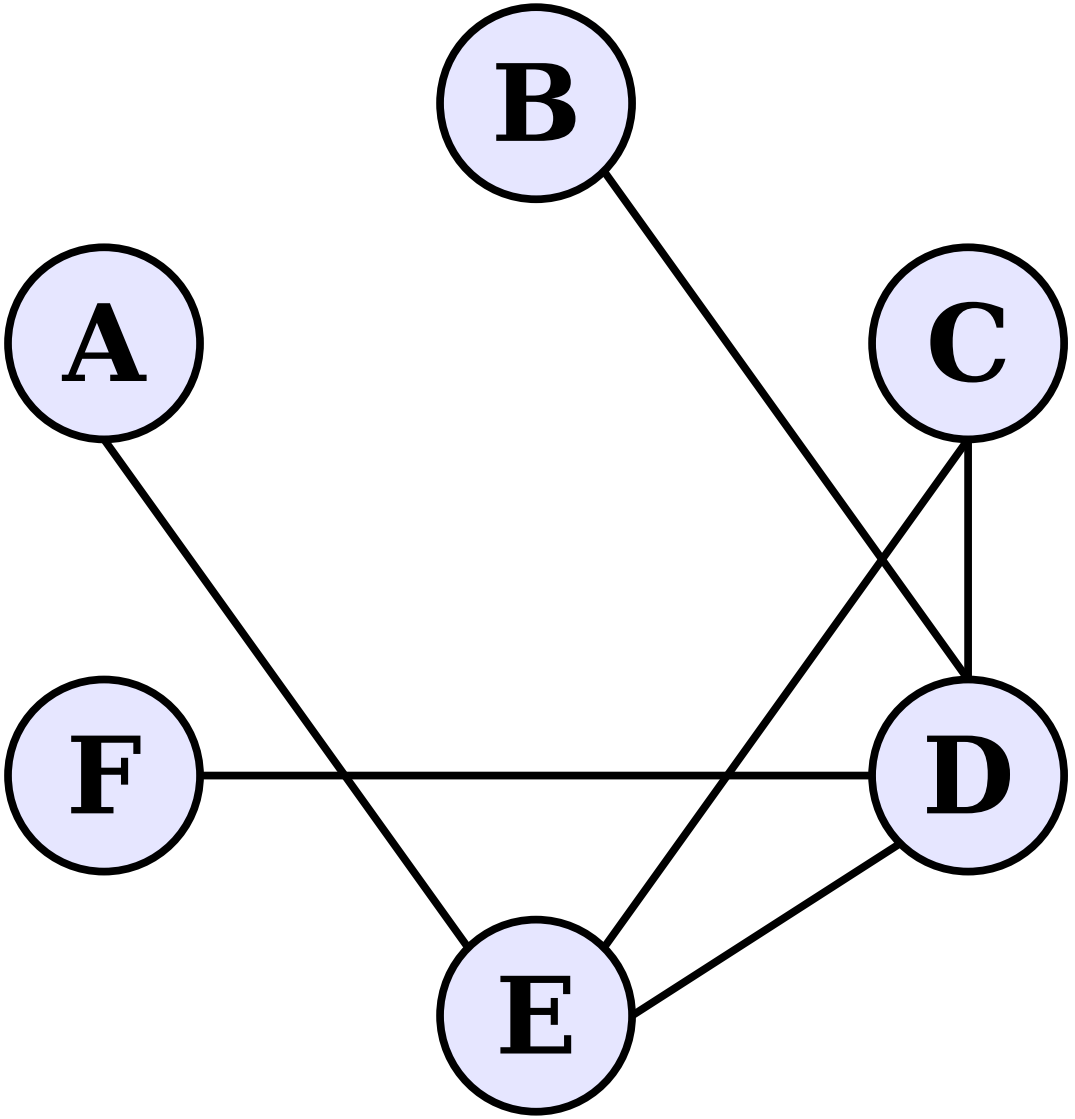
It's the previous ***patience sorting*** algorithm, with some clever implementation tricks.

This algorithm works by exploiting particular aspects of how longest increasing subsequences are constructed. It's not immediately obvious that it works correctly.

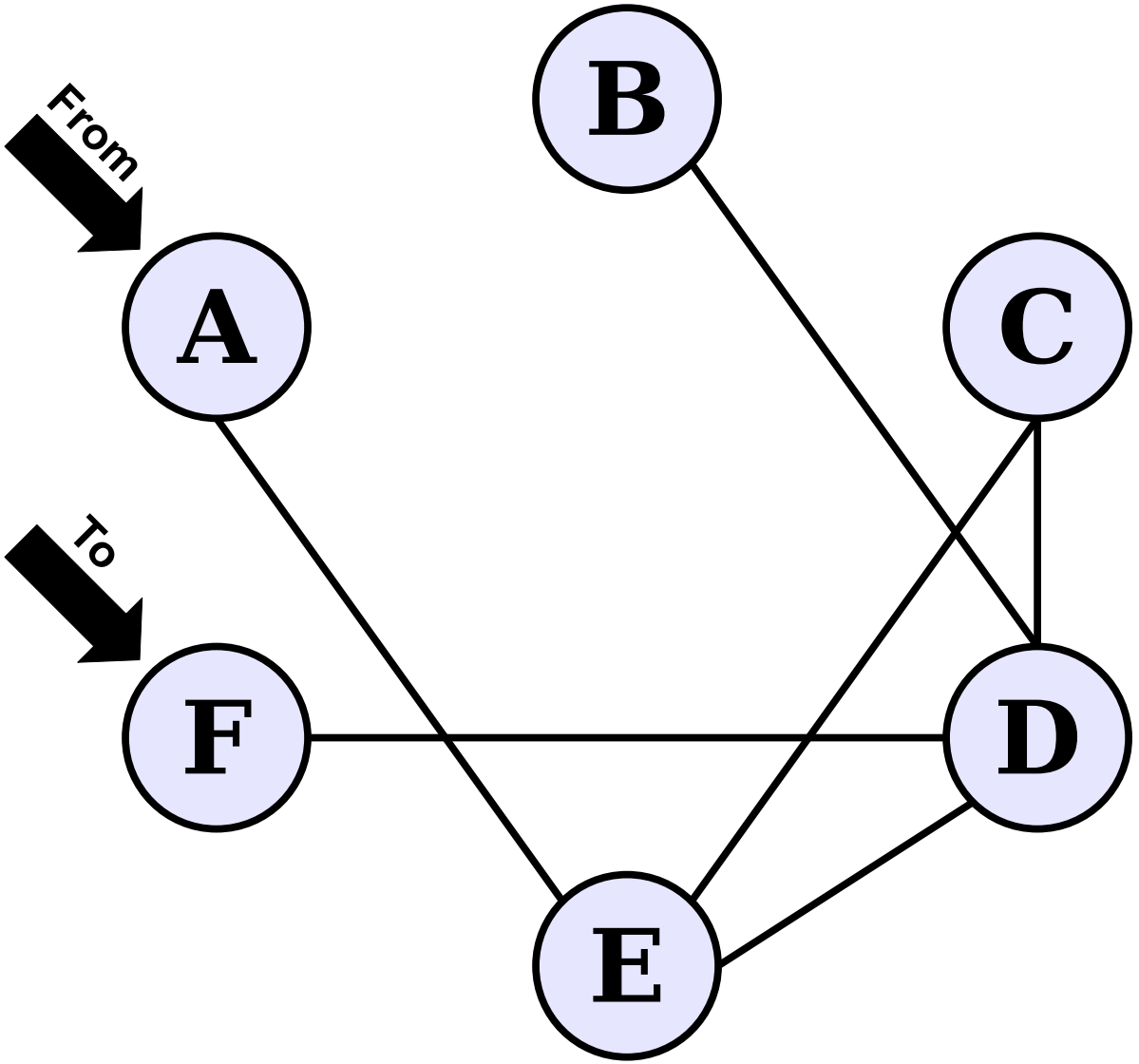
Phenomenal Exercise 1: Prove that this procedure always works!

Phenomenal Exercise 2: Show that you can actually implement this same algorithm in time $O(n \log n)$.

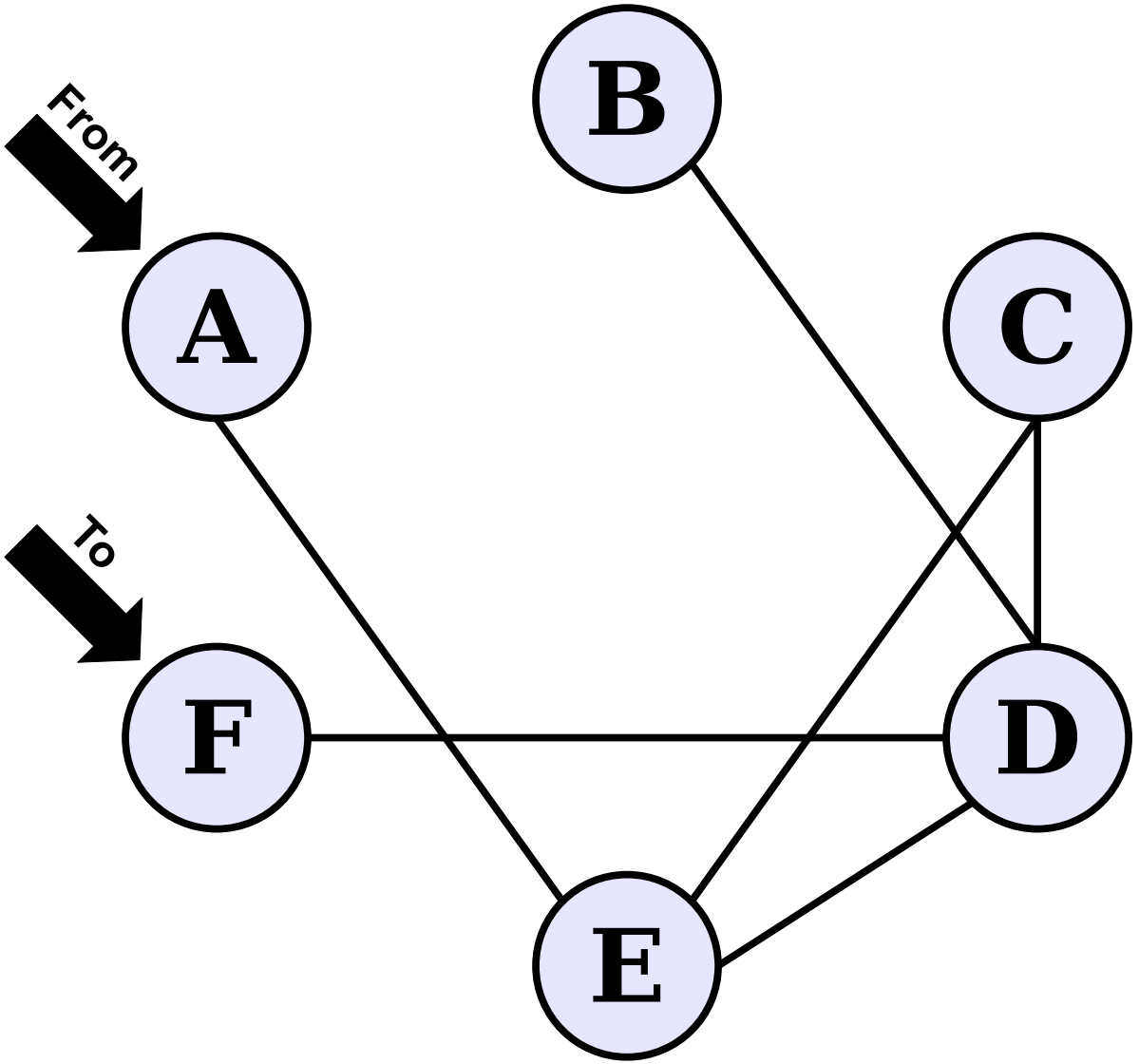
Another Problem



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 $O(m + n)$.

Proof idea: Use breadth-first search!

The algorithm is a bit nuanced. It uses some specific properties of shortest paths and the proof of correctness is 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, 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**.

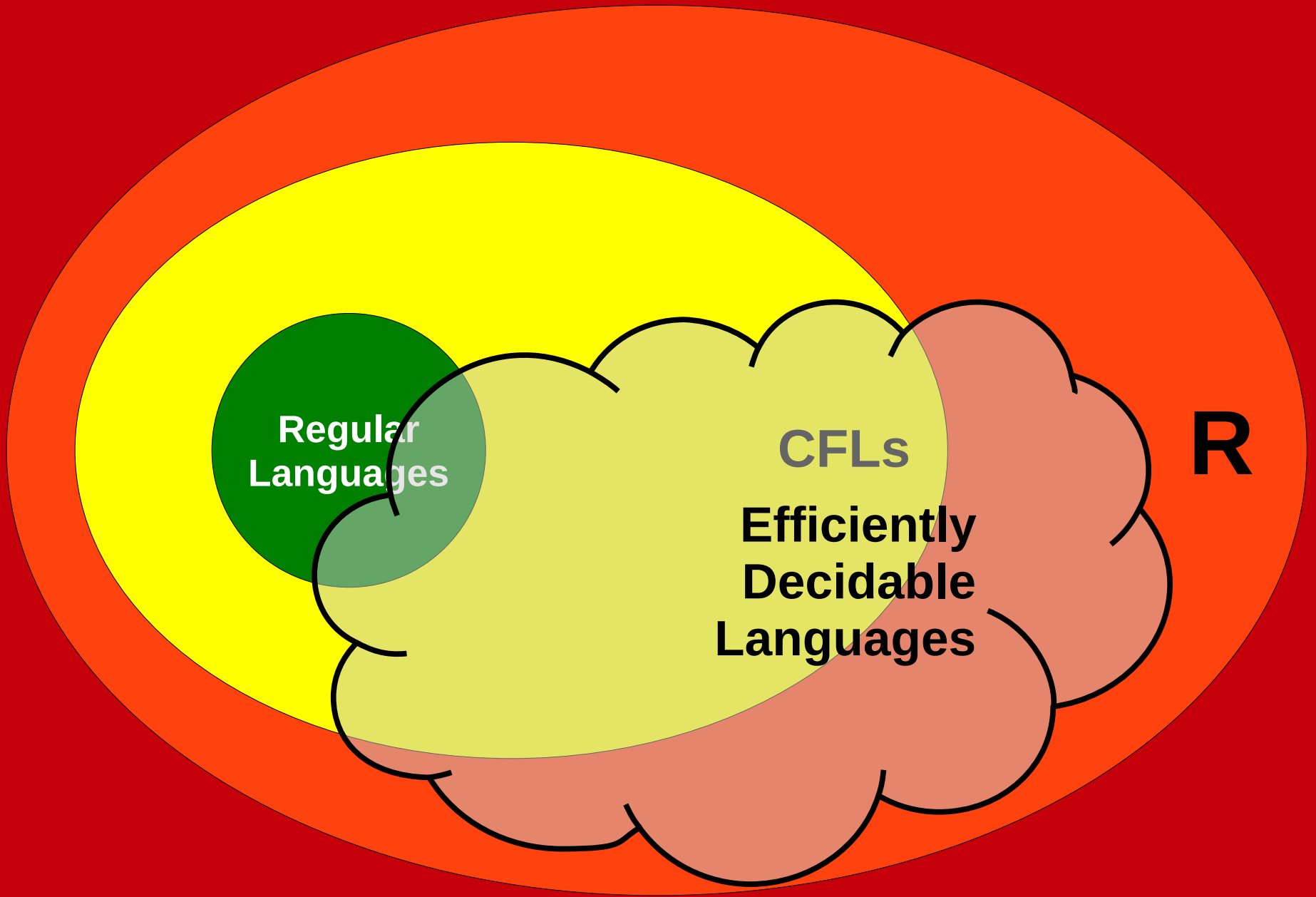
- All have linear-time TMs.

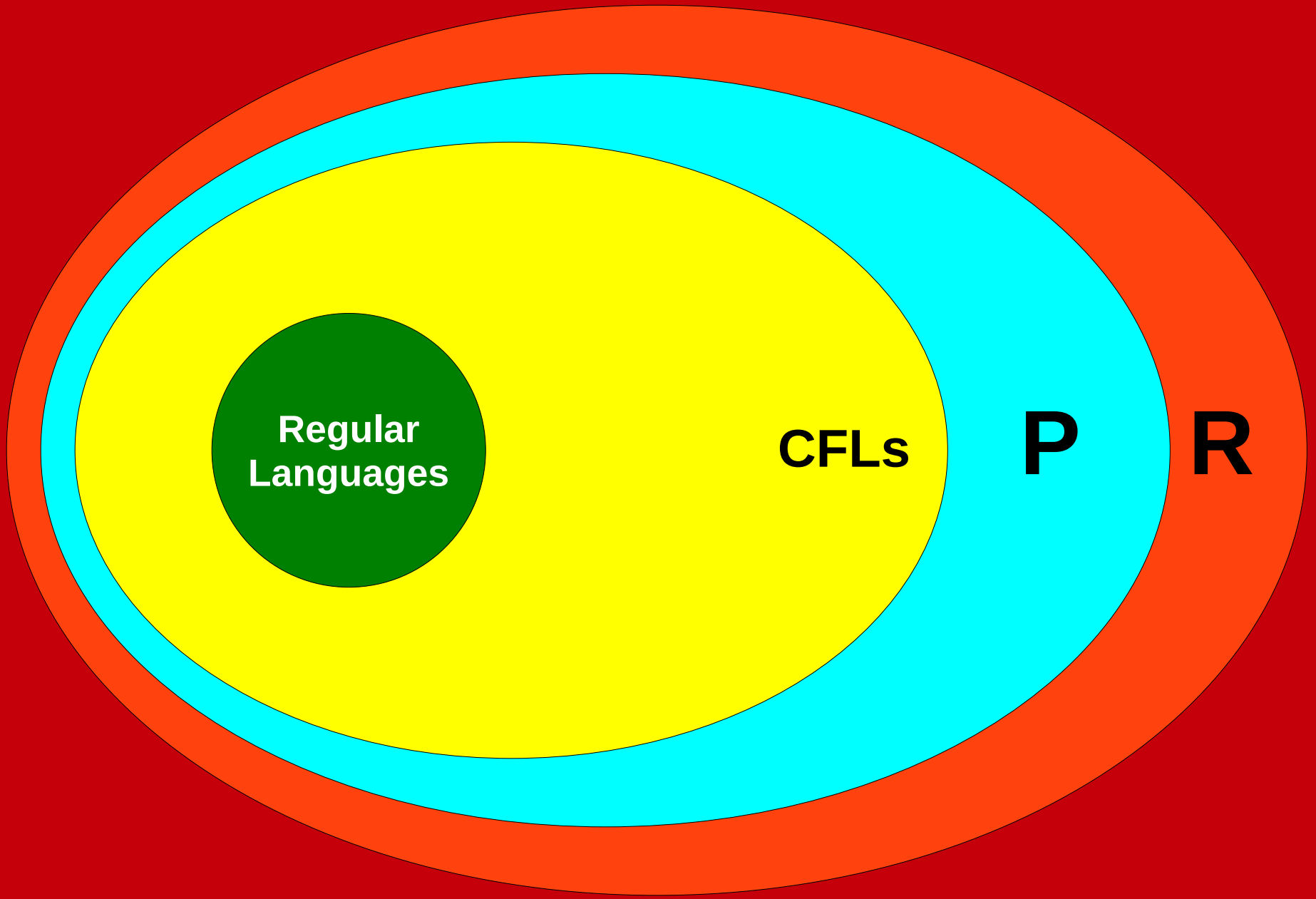
All CFLs are in **P**.

- Requires a more nuanced argument (the *CYK algorithm* or *Earley's algorithm*.)

And a *ton* of other problems are in **P** as well.

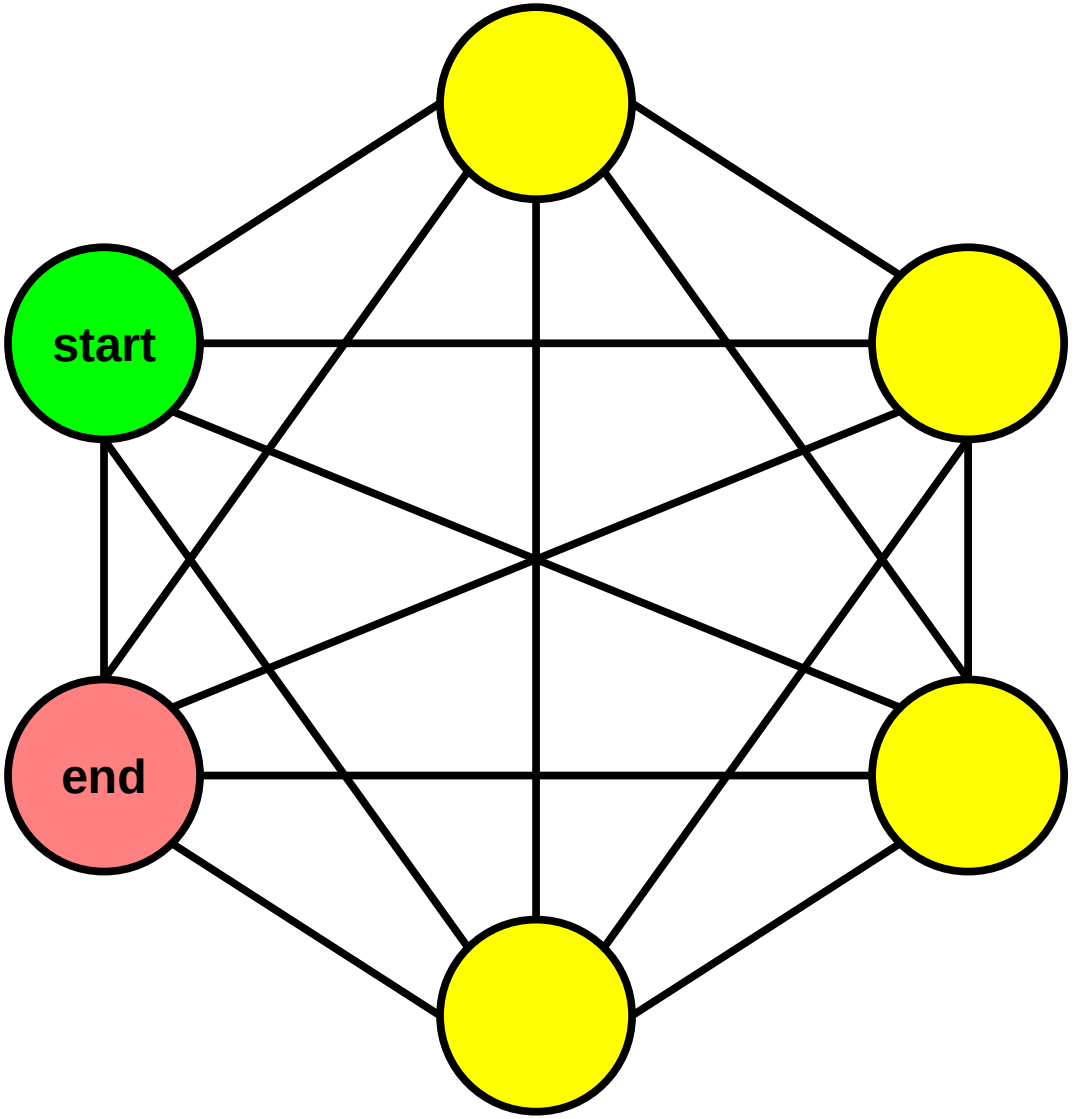
Curious? Take CS161!


Regular Languages
CFLs
Efficiently Decidable Languages
R
Undecidable Languages



Undecidable Languages

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 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

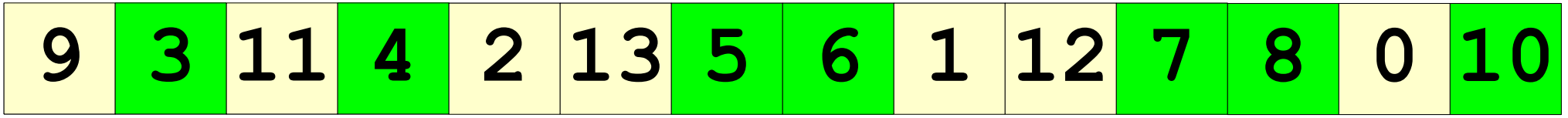
Does this Sudoku problem have a solution?

Verifiers – Again

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

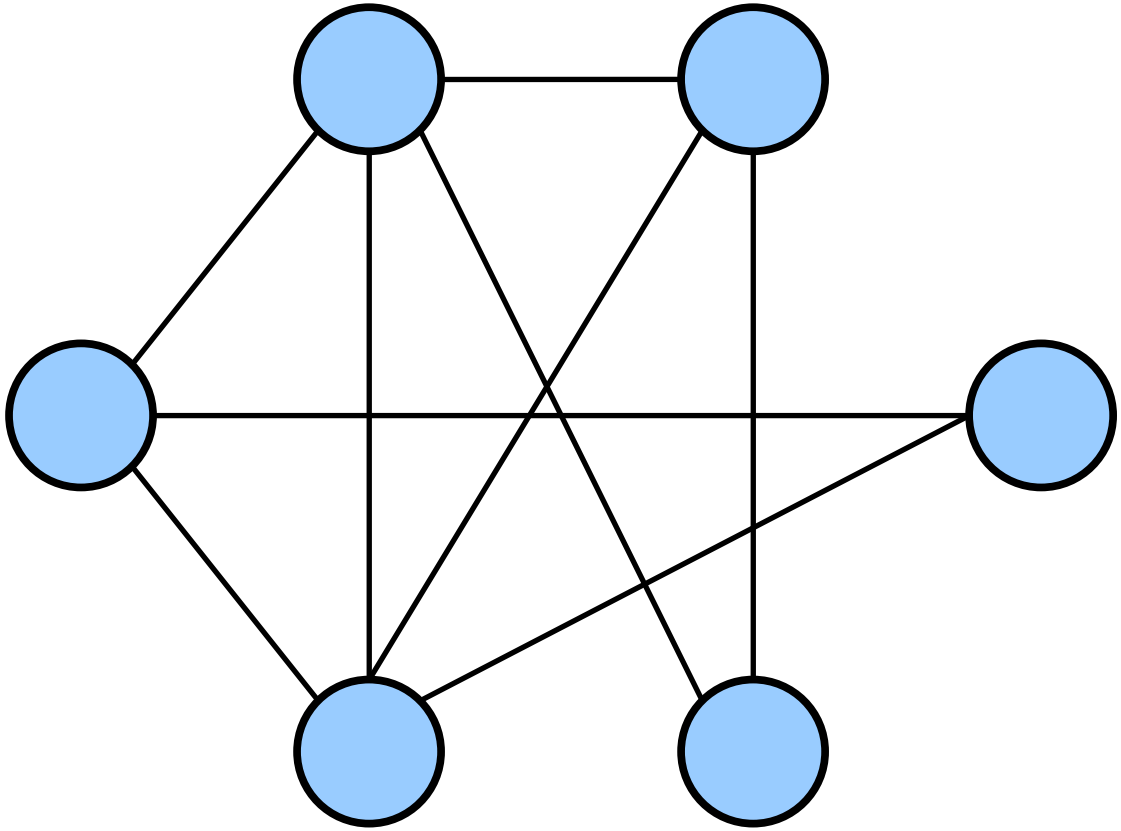
Is there an ascending subsequence of length at least 7?

Verifiers – Again



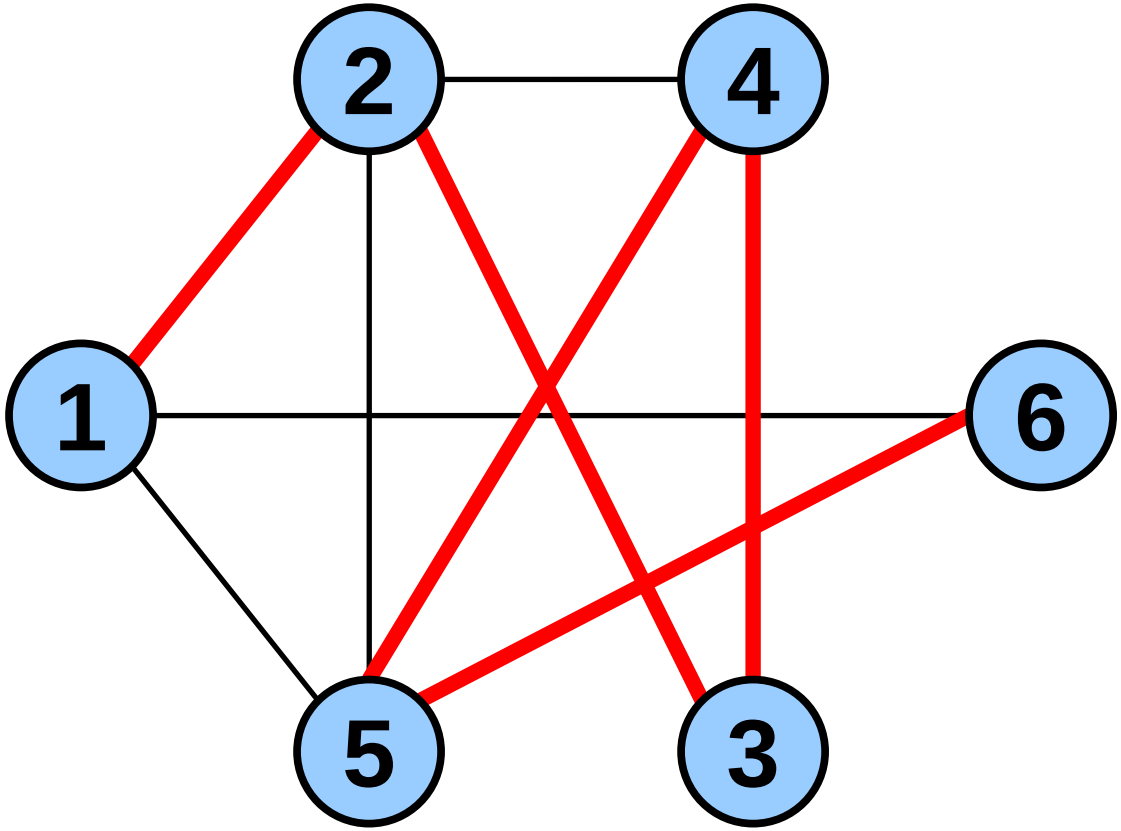
Is there an ascending subsequence of length at least 7?

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.

Useful fact: $\mathbf{NP} \subsetneq \mathbf{R}$. Come talk to me after class if you're curious why!

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

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

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

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

We know that **R ≠ RE**.

So does that mean **P ≠ NP**?

And now...

The

Biggest Question

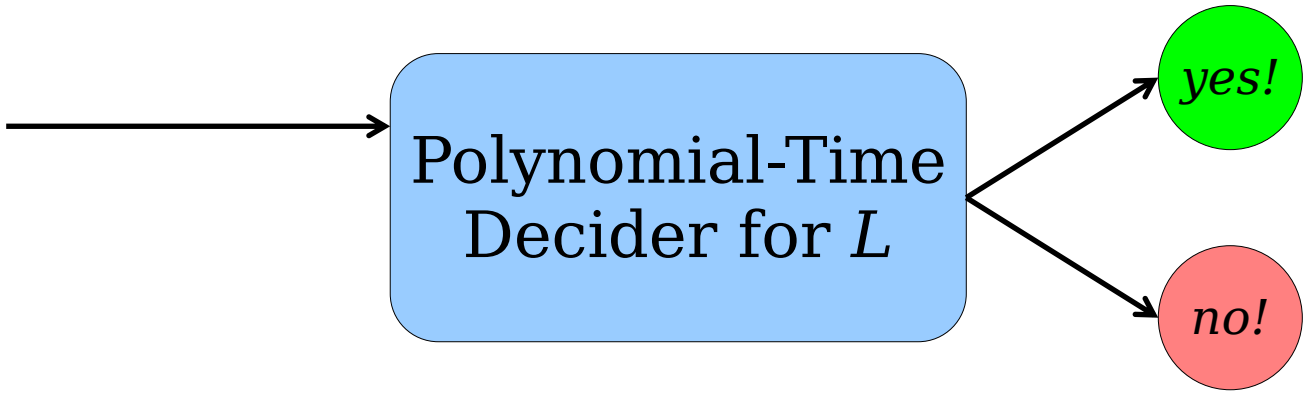
in

Theoretical Computer Science

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

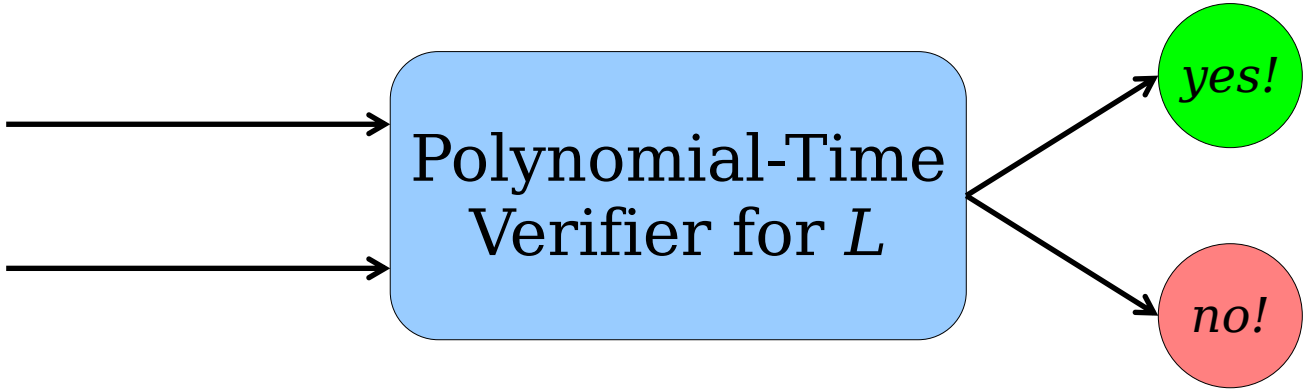
P = { L | There is a polynomial-time decider for L }

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



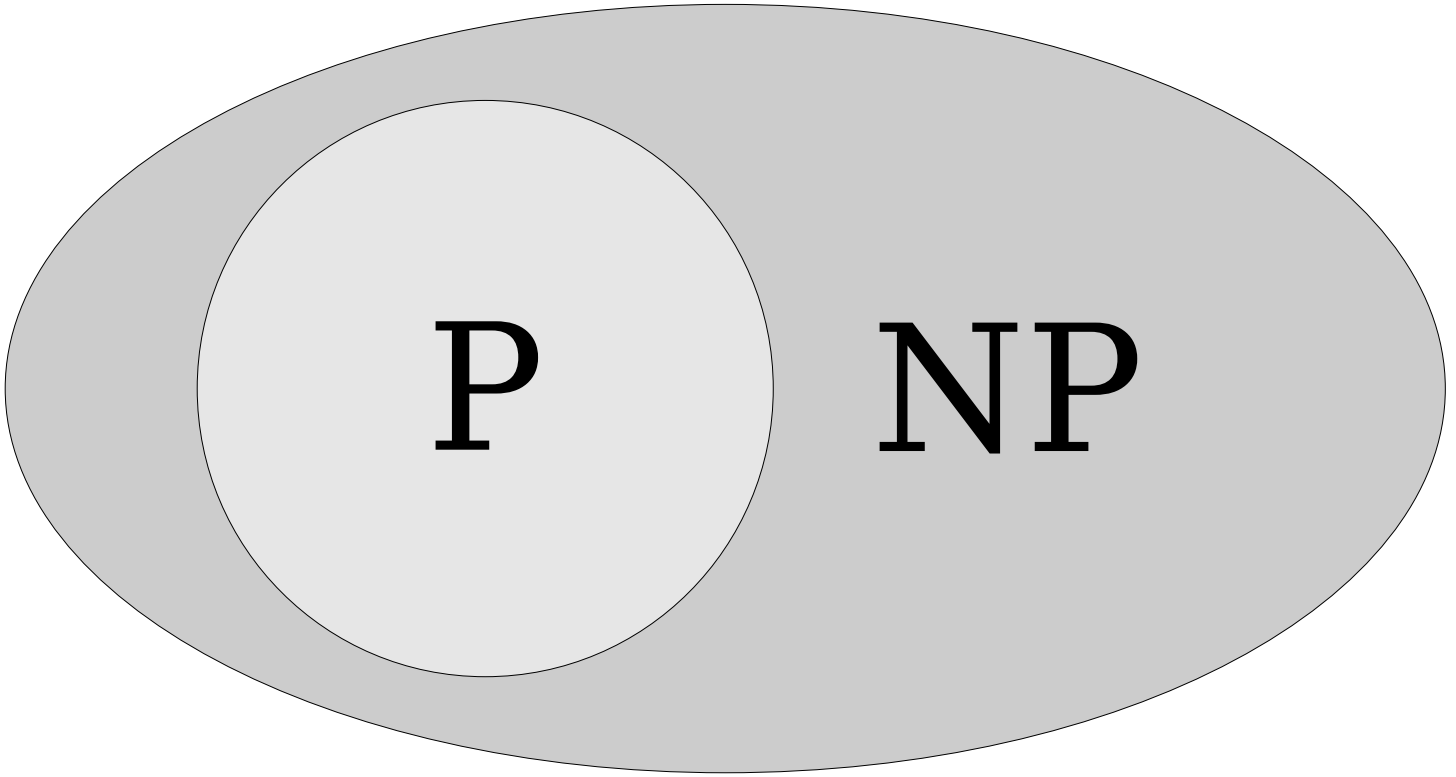
P = { L | There is a polynomial-time decider for L }

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

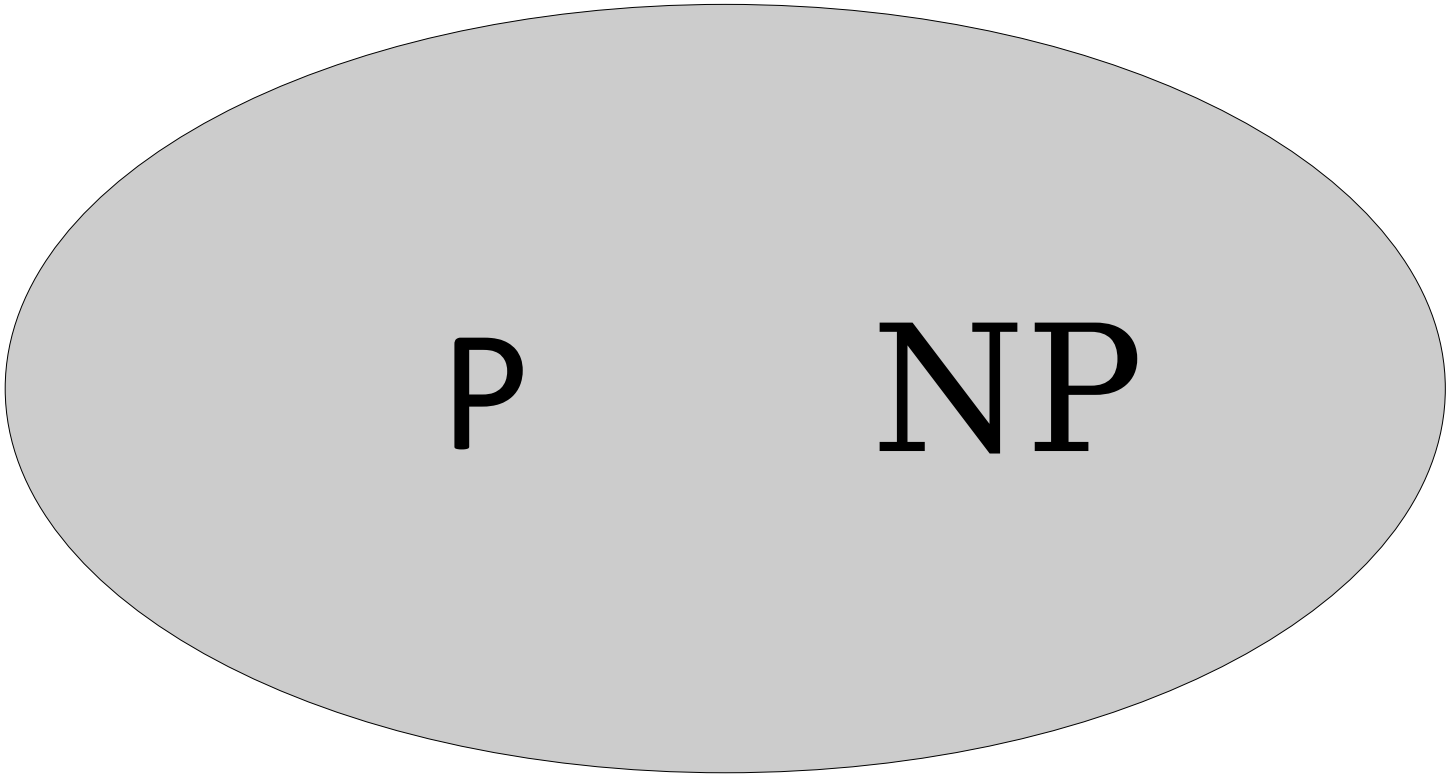


P ⊆ NP

Which Picture is Correct?



Which Picture is Correct?



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

The $\mathbf{P \stackrel{?}{=} NP}$ 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 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 **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**.

“My hunch is that [**P** $\stackrel{?}{=}$ **NP**] will be solved by a young researcher who is not encumbered by too much conventional wisdom about how to attack the problem.”

– Prof. Richard Karp

(The guy who first popularized the* P $\stackrel{?}{=}$ NP *problem.)

“There is something very strange about this problem, something very philosophical. It is the greatest unsolved problem in mathematics [...] It is the *raison d’être* of abstract computer science, and as long as it remains unsolved, its mystery will ennoble the field.”

-Prof. Jim Owings

(Computability/Complexity theorist)

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

Adapting our Techniques

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

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

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

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

We know that **R ≠ RE**.

So does that mean **P ≠ NP**?

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.

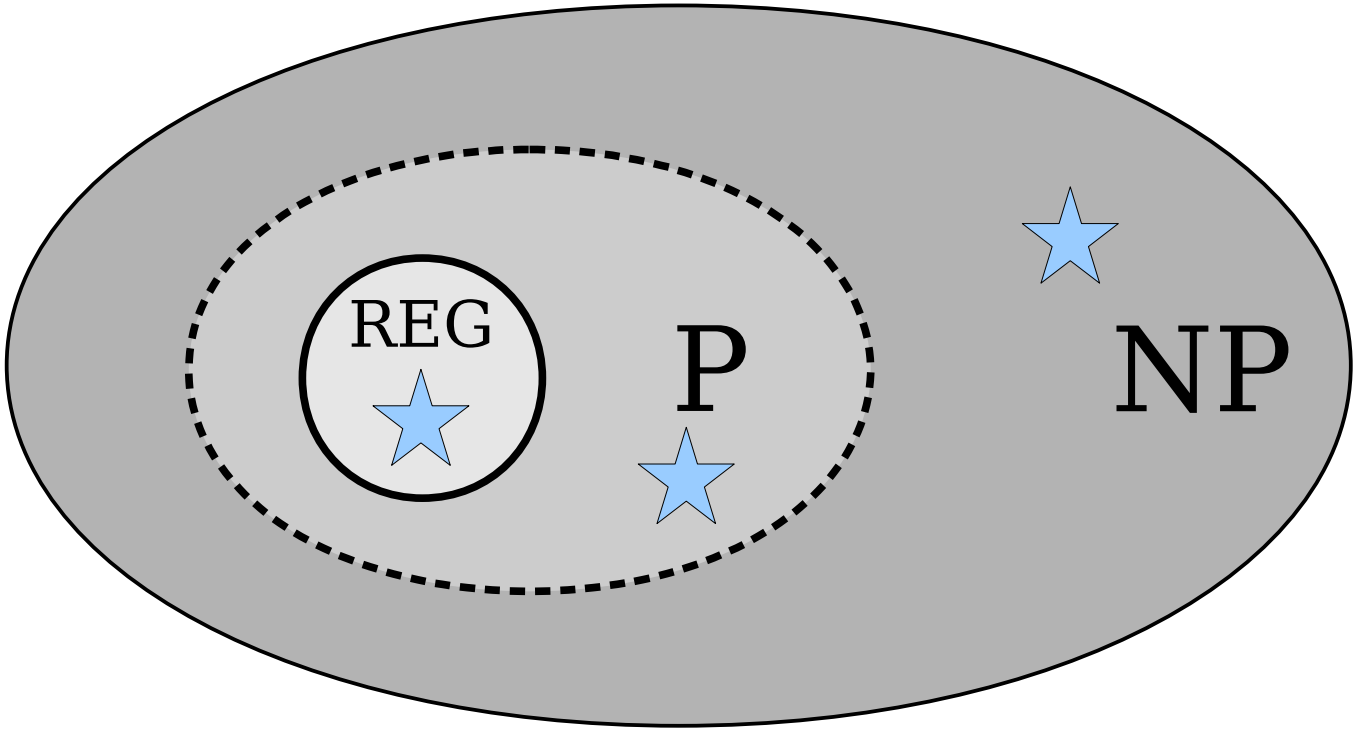
Self-Reference: TMs can get their own source code.

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

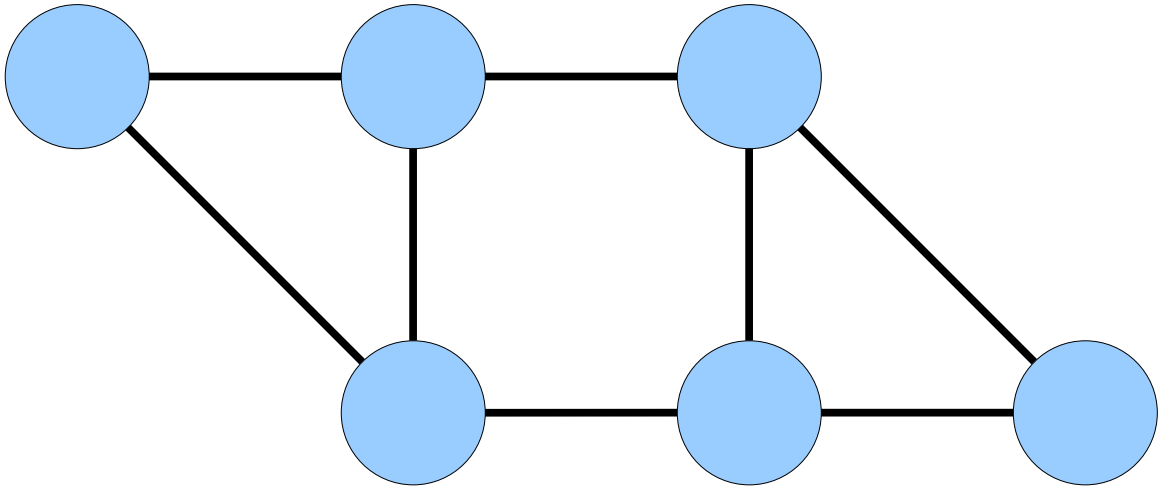
Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.

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.

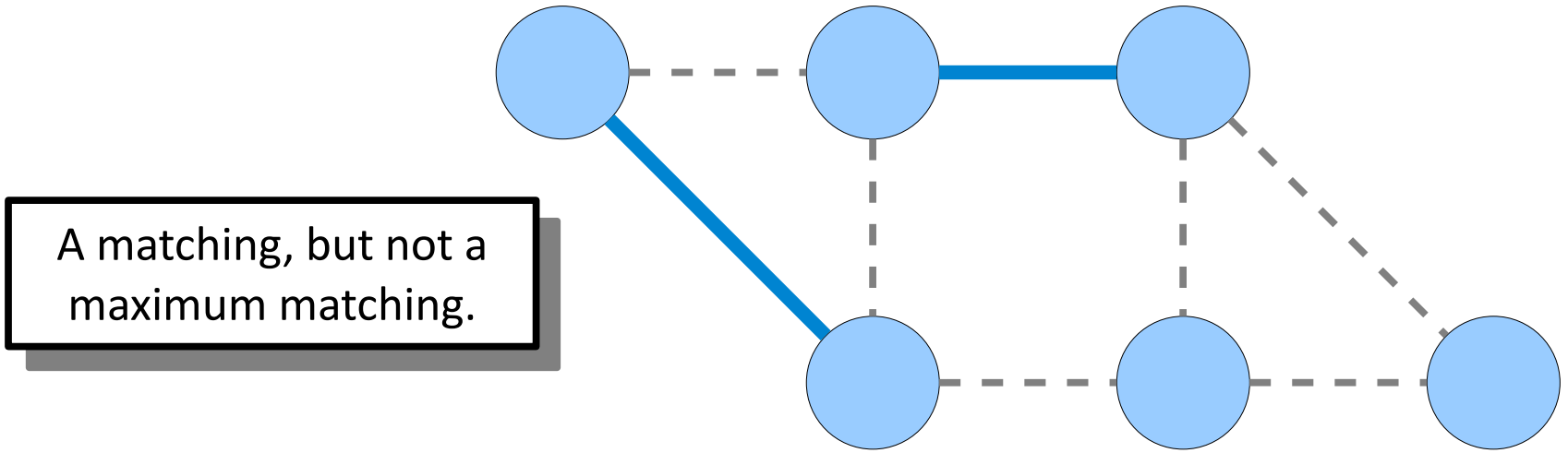
A ***maximum matching*** is a matching with the largest number of edges.



Maximum Matching

Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.

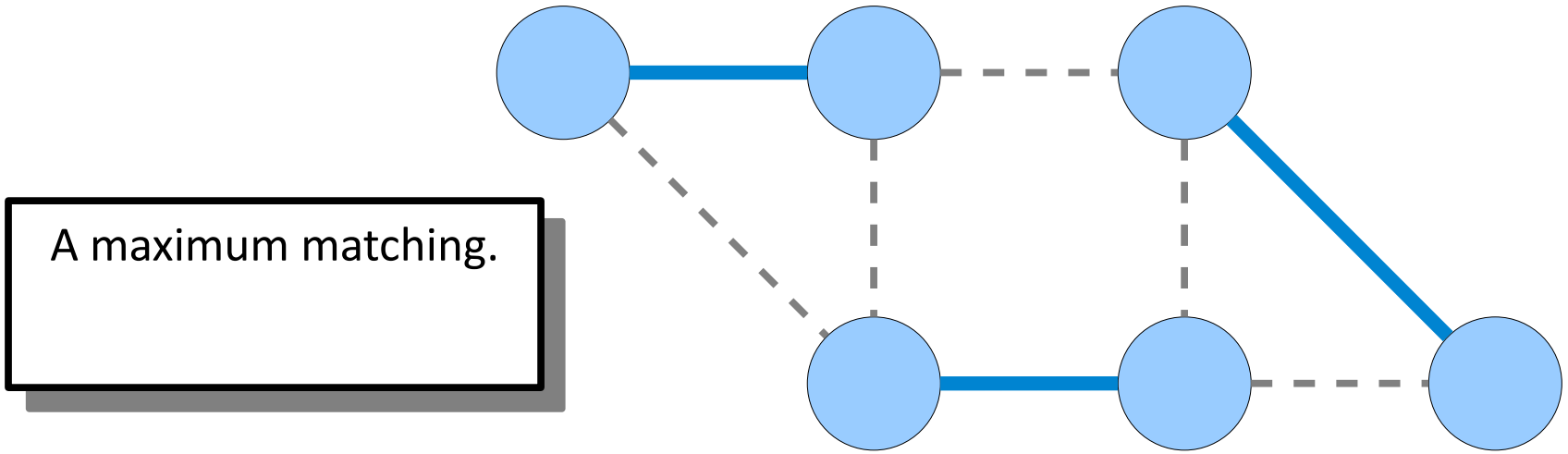
A ***maximum matching*** is a matching with the largest number of edges.



Maximum Matching

Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.

A ***maximum matching*** is a matching with the largest number of edges.

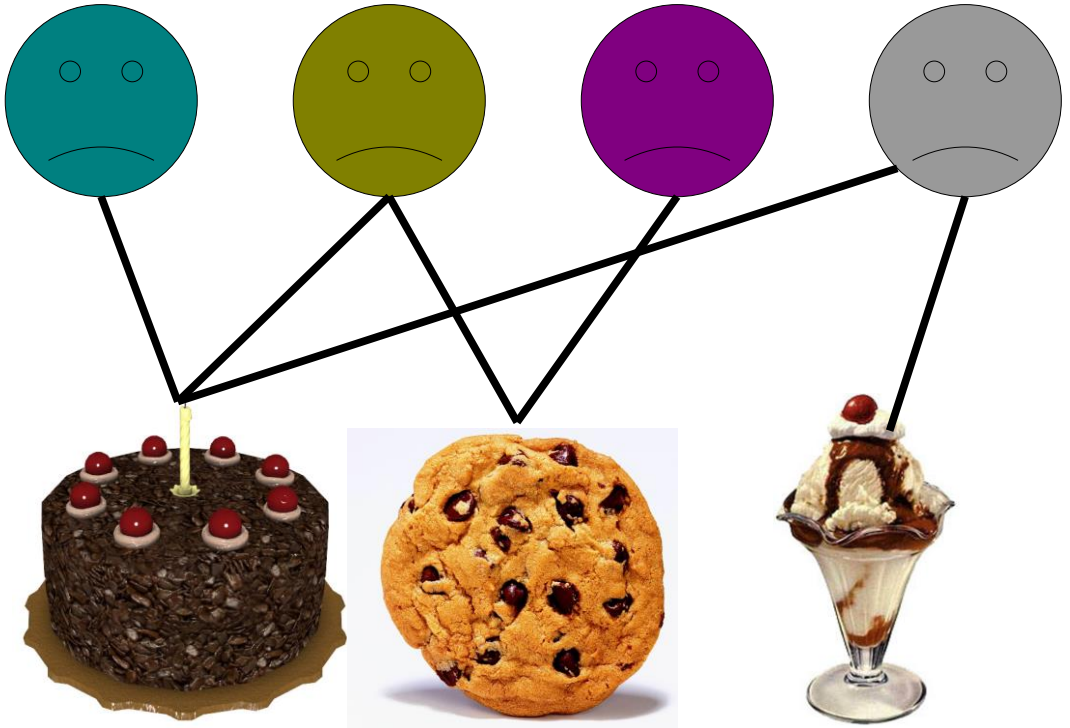


A maximum matching.

Maximum Matching

Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.

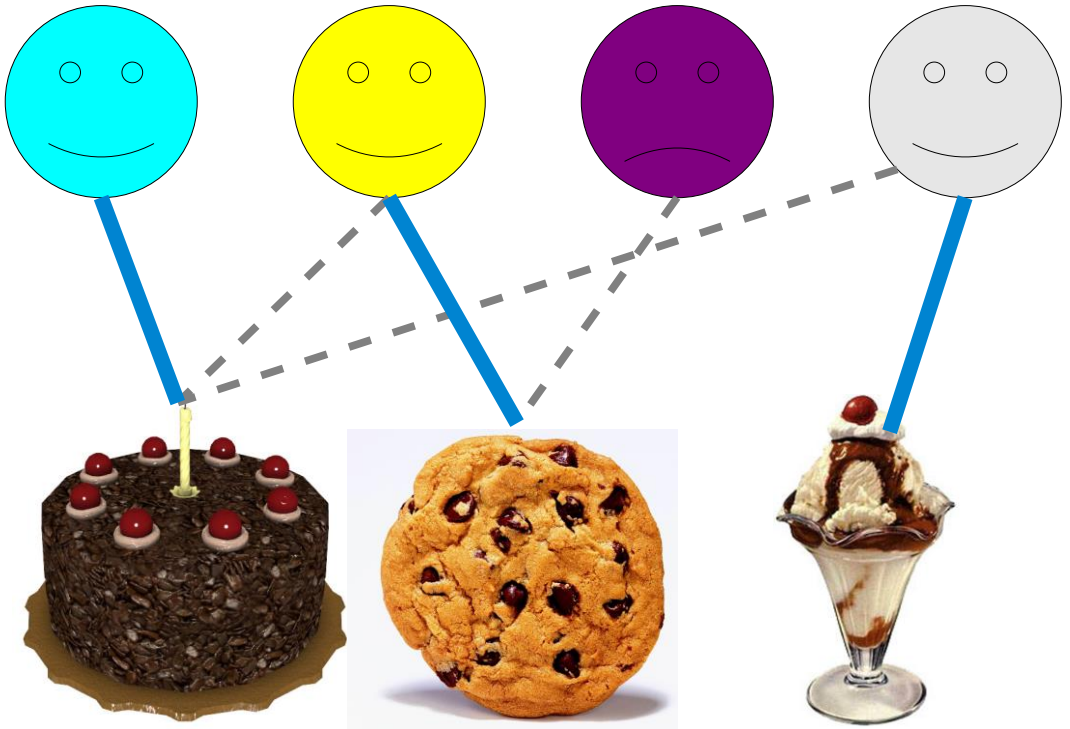
A ***maximum matching*** is a matching with the largest number of edges.



Maximum Matching

Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.

A ***maximum matching*** is a matching with the largest number of edges.



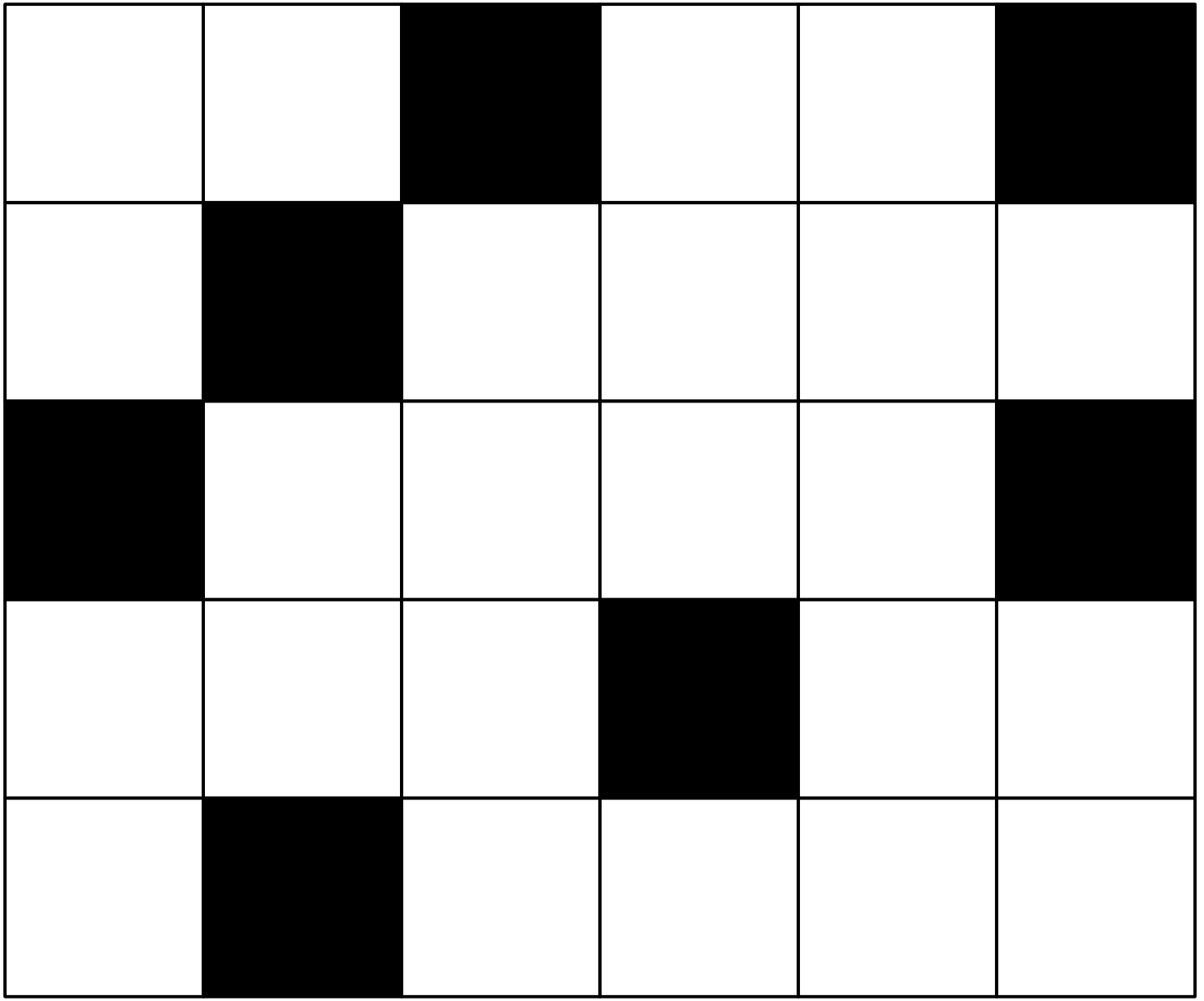
Maximum Matching

Jack Edmonds' paper “Paths, Trees, and Flowers” gives a polynomial-time algorithm for finding maximum matchings.

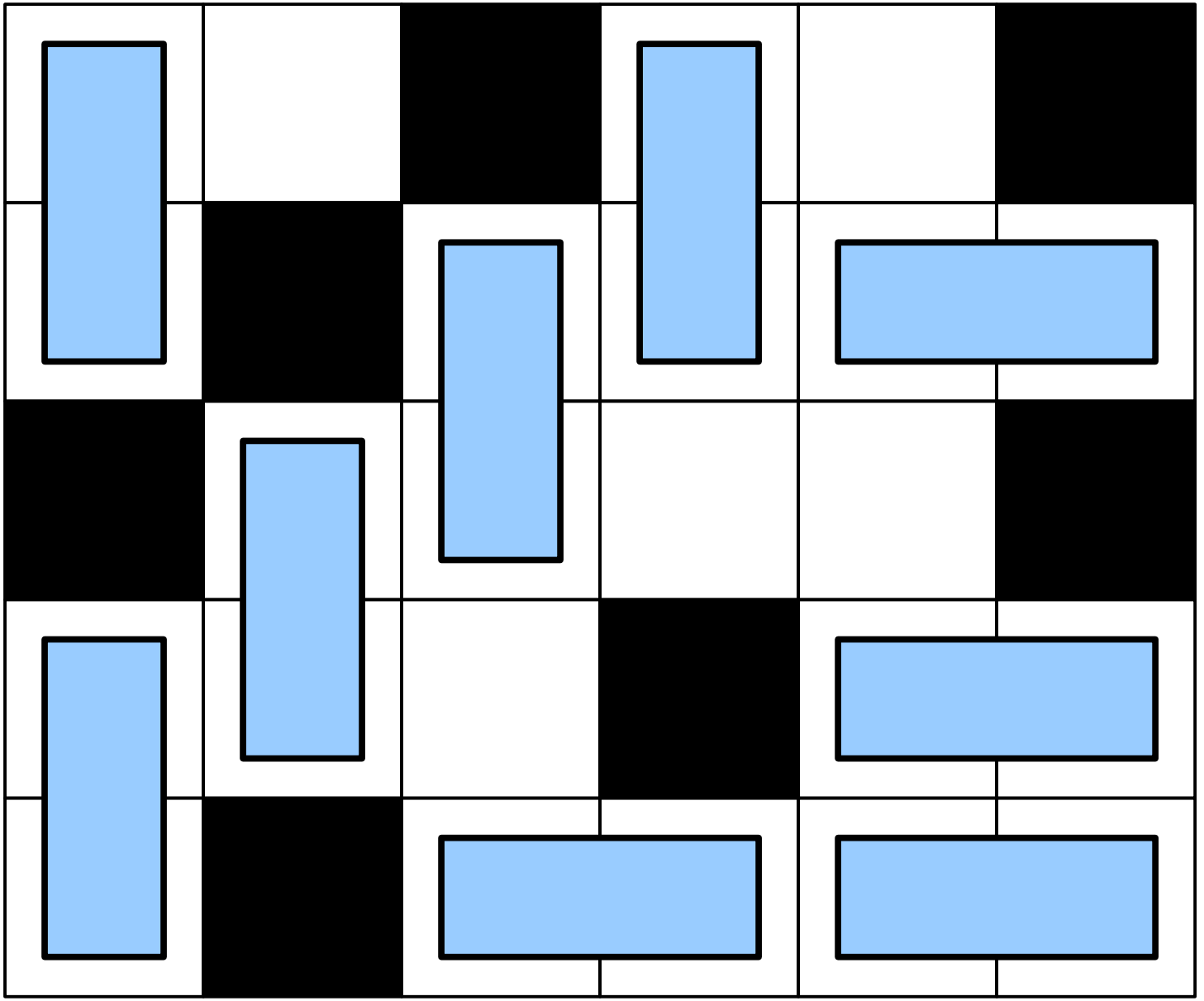
He’s the guy with the quote about “better than decidable.”

Using this fact, what other problems can we solve?

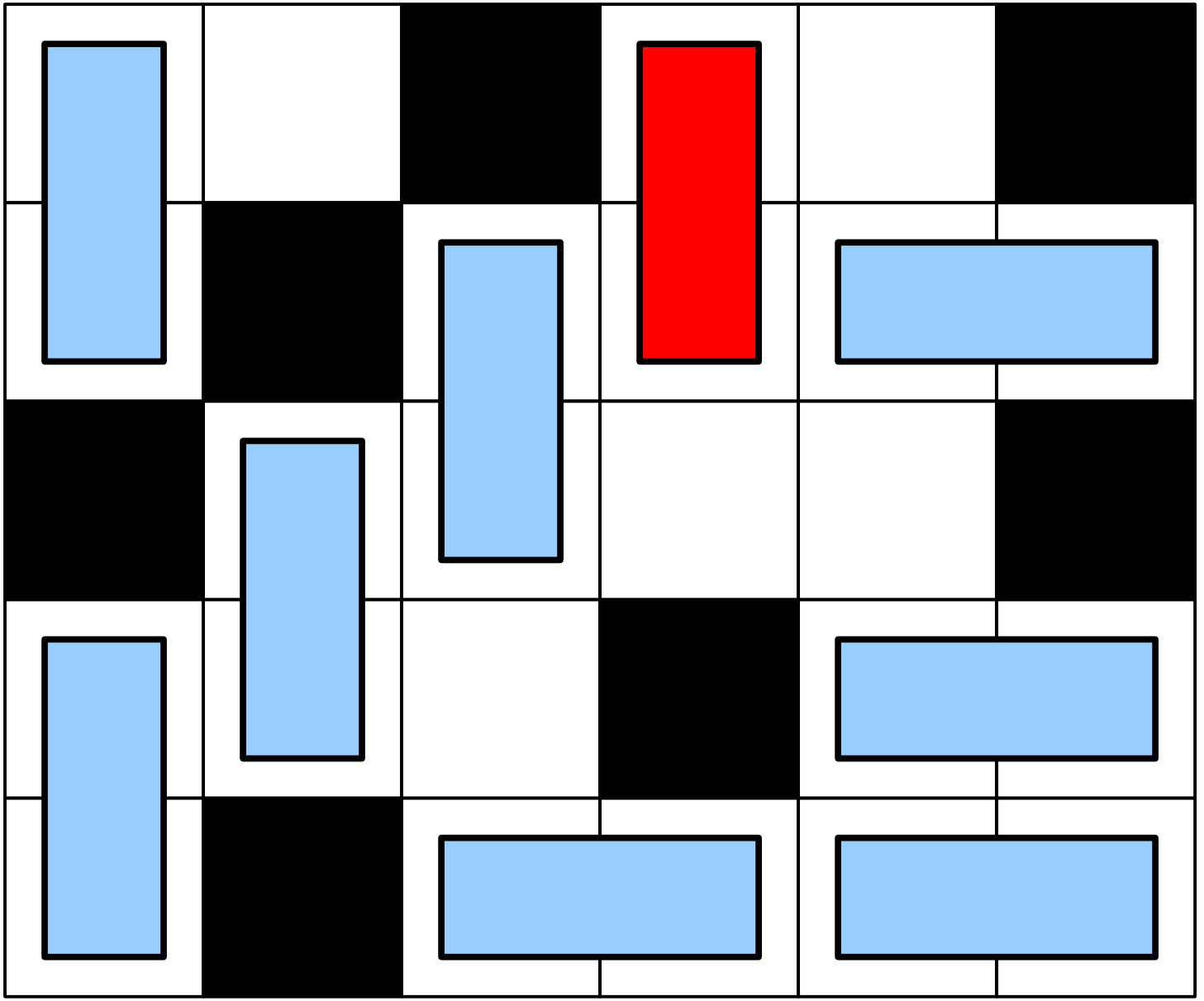
Domino Tiling



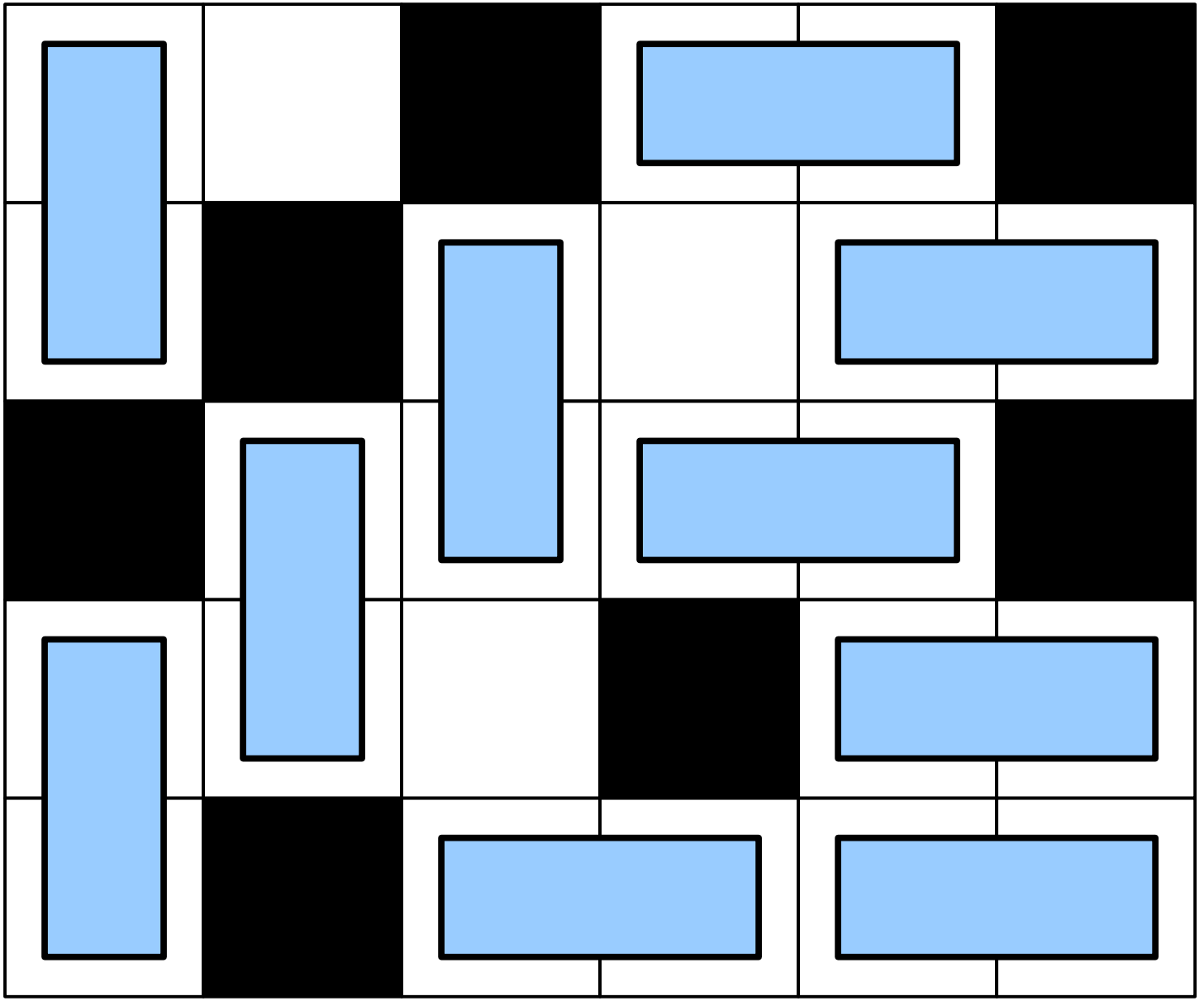
Domino Tiling



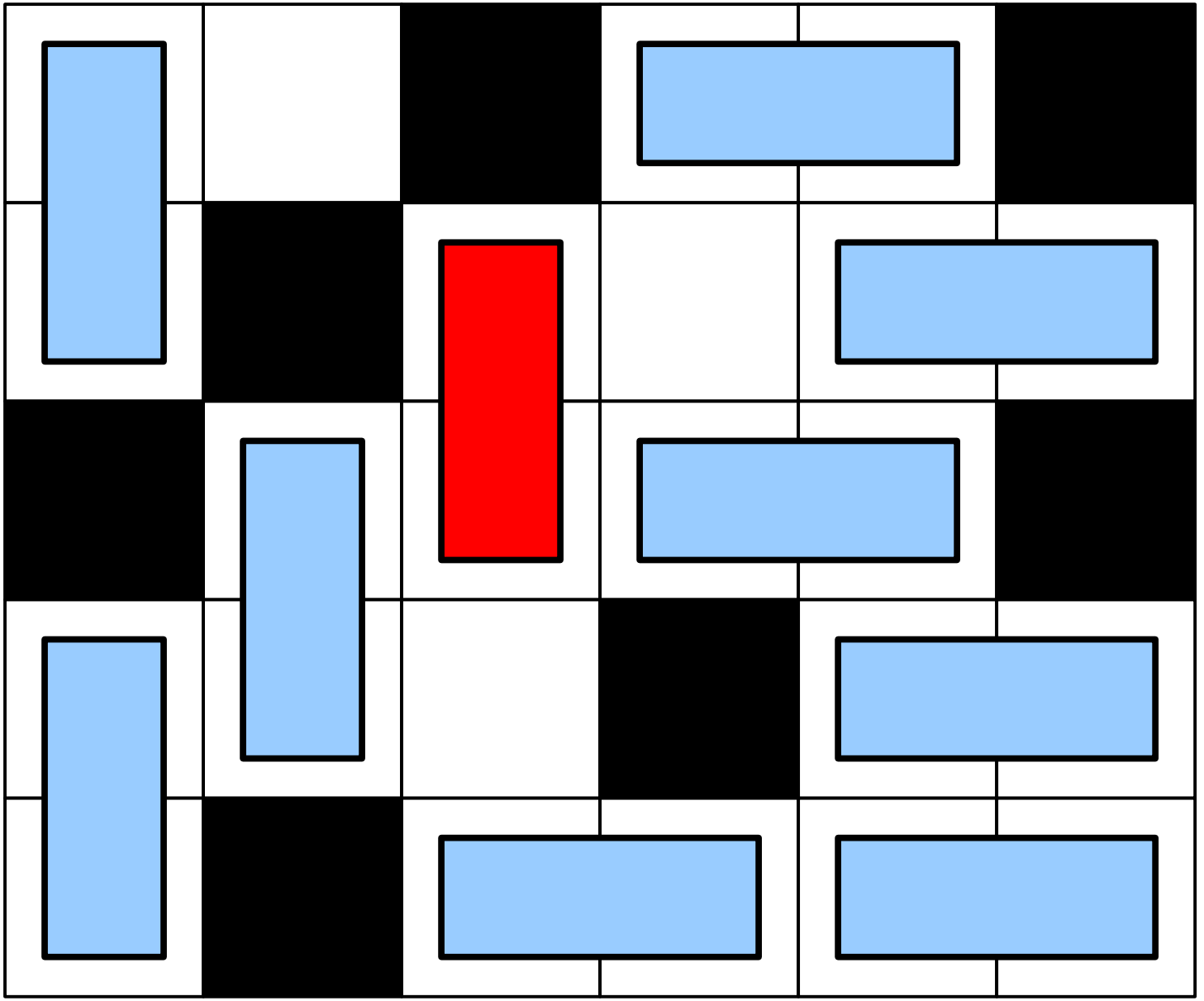
Domino Tiling



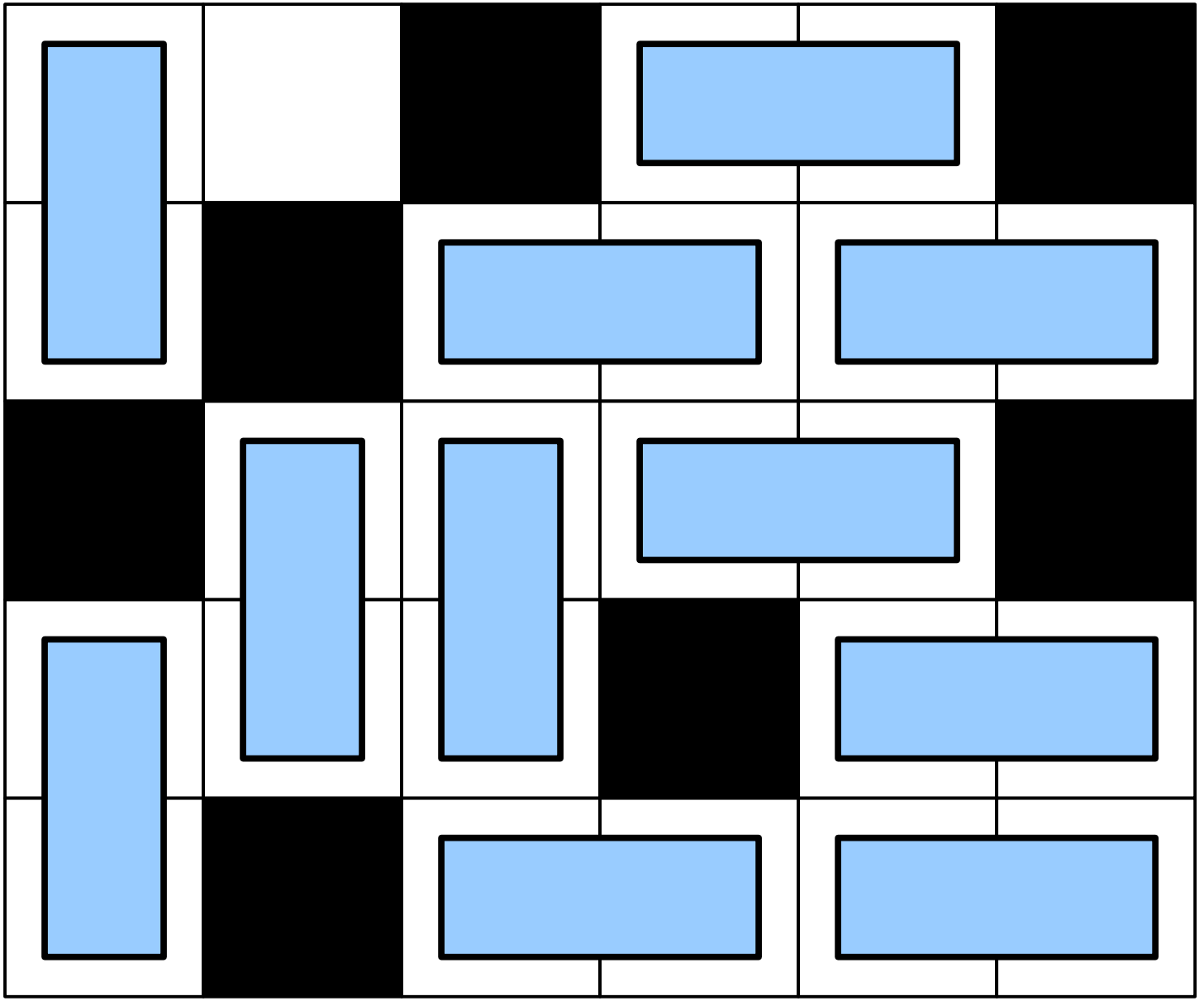
Domino Tiling



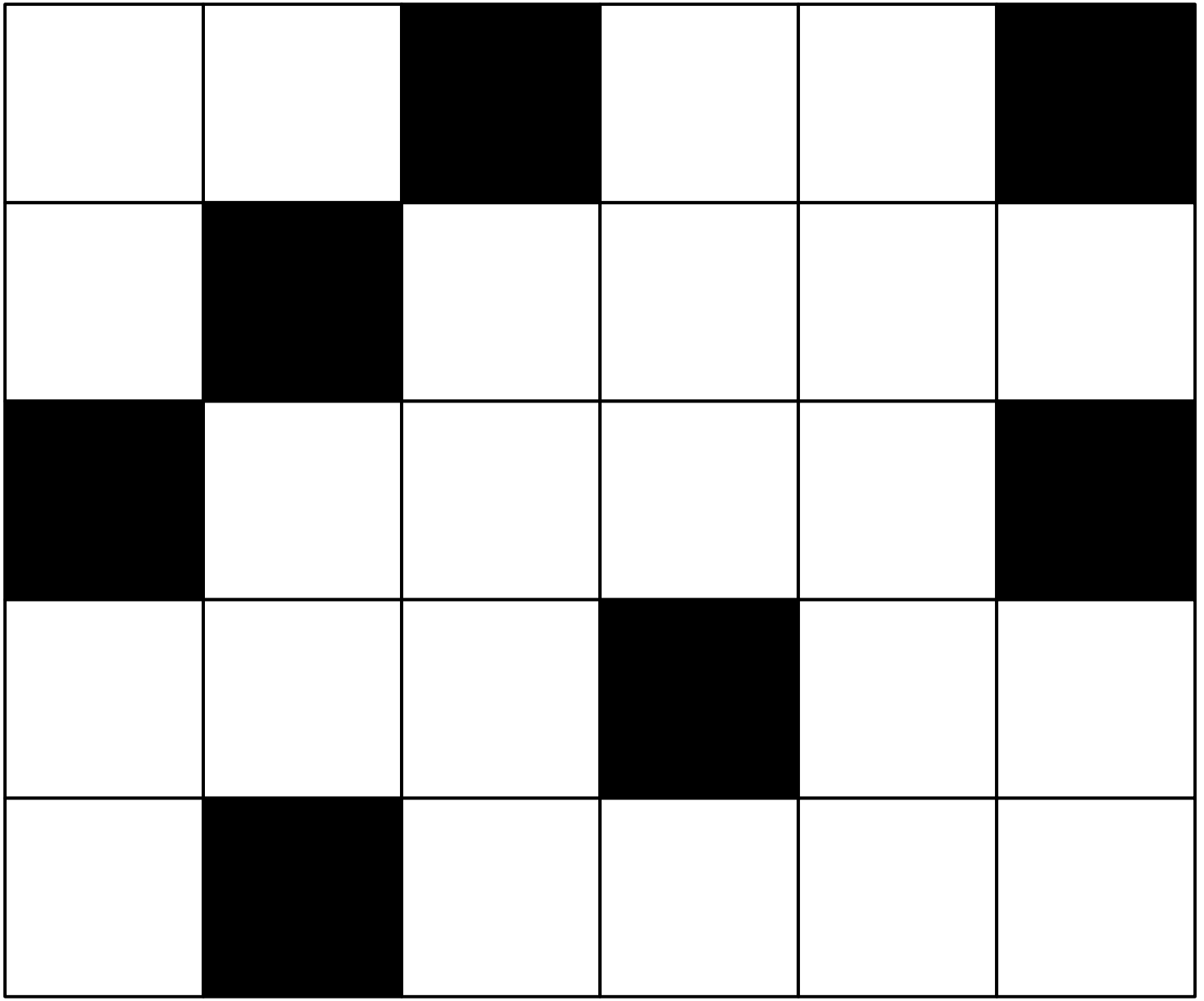
Domino Tiling



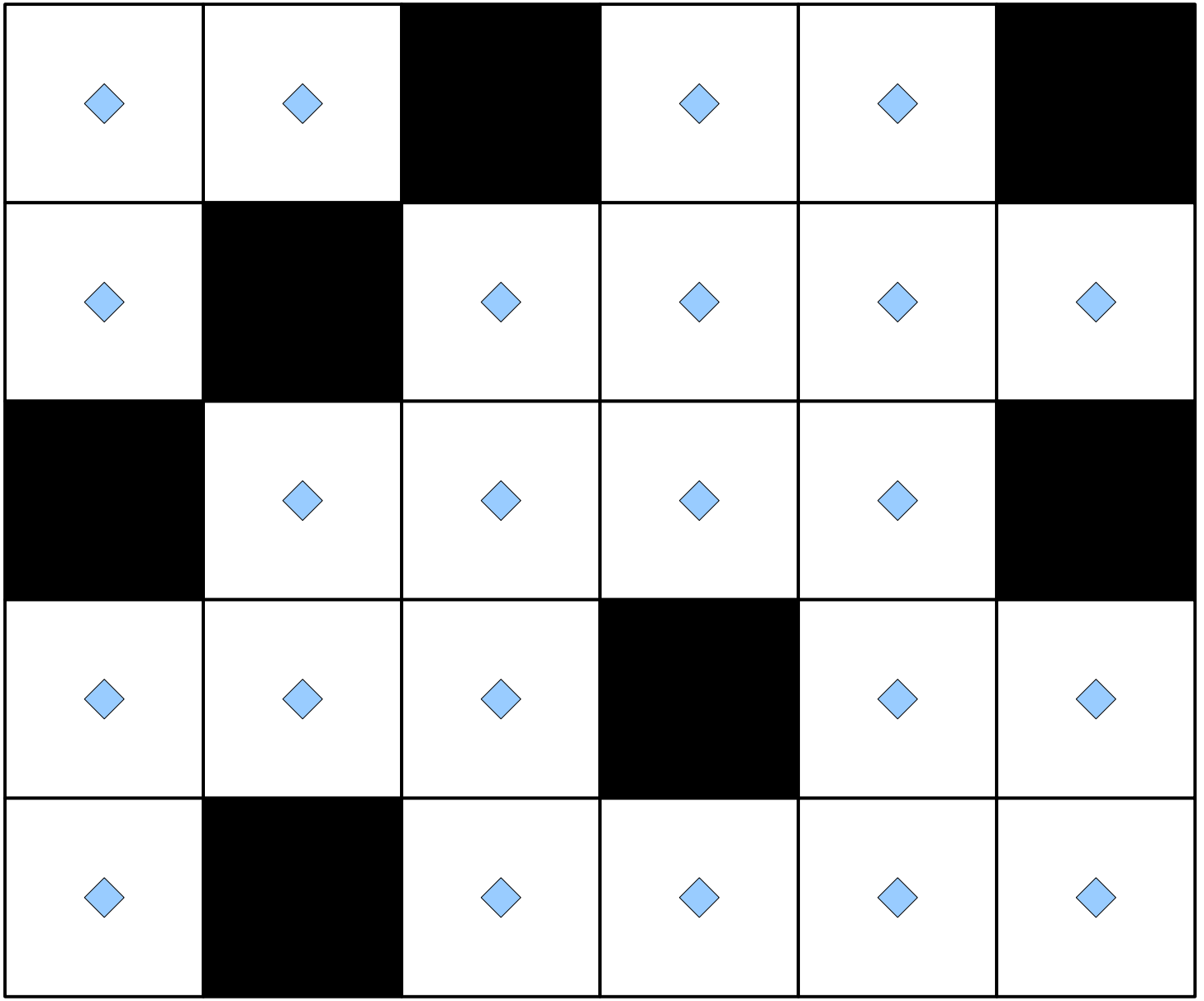
Domino Tiling



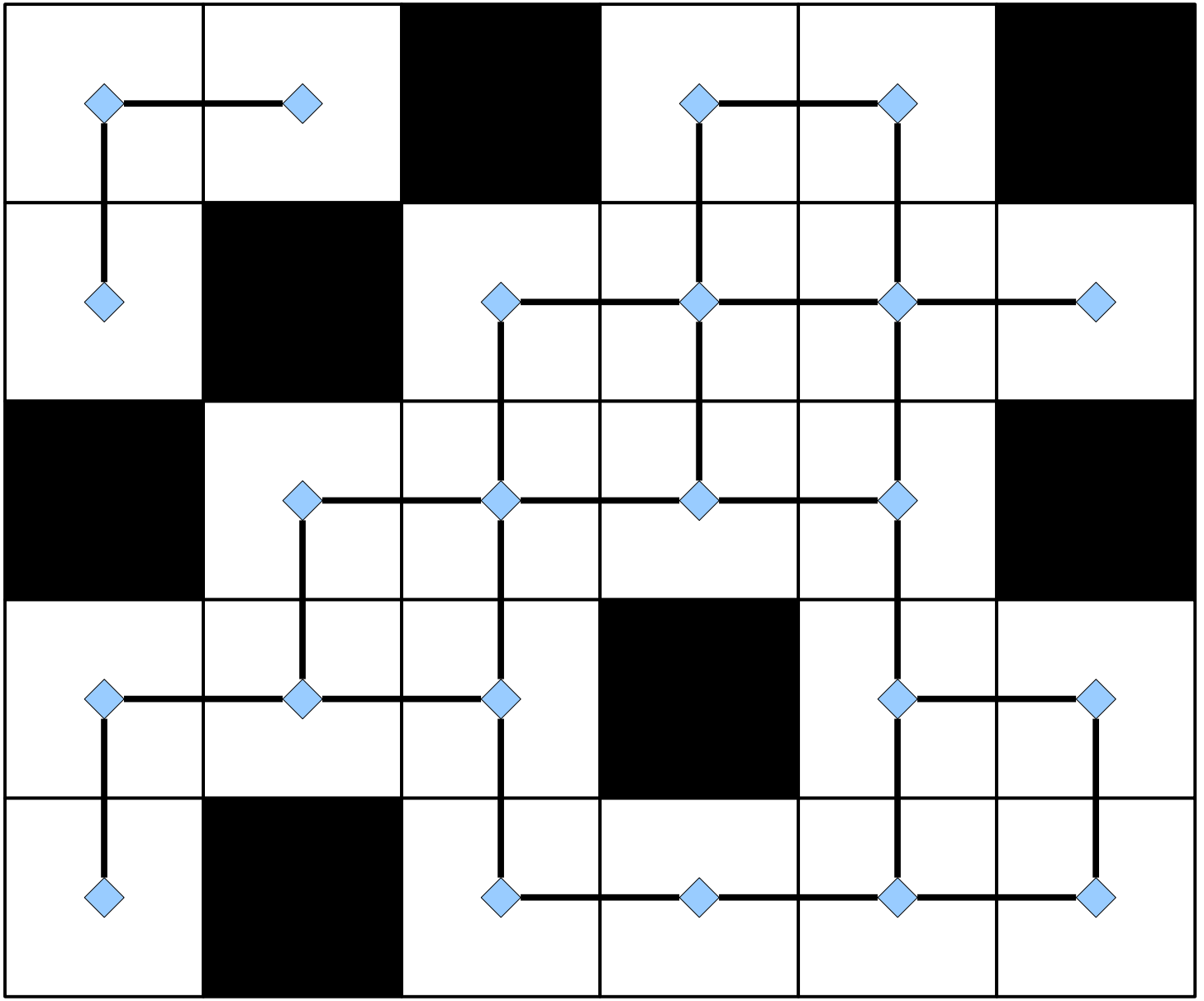
Solving Domino Tiling



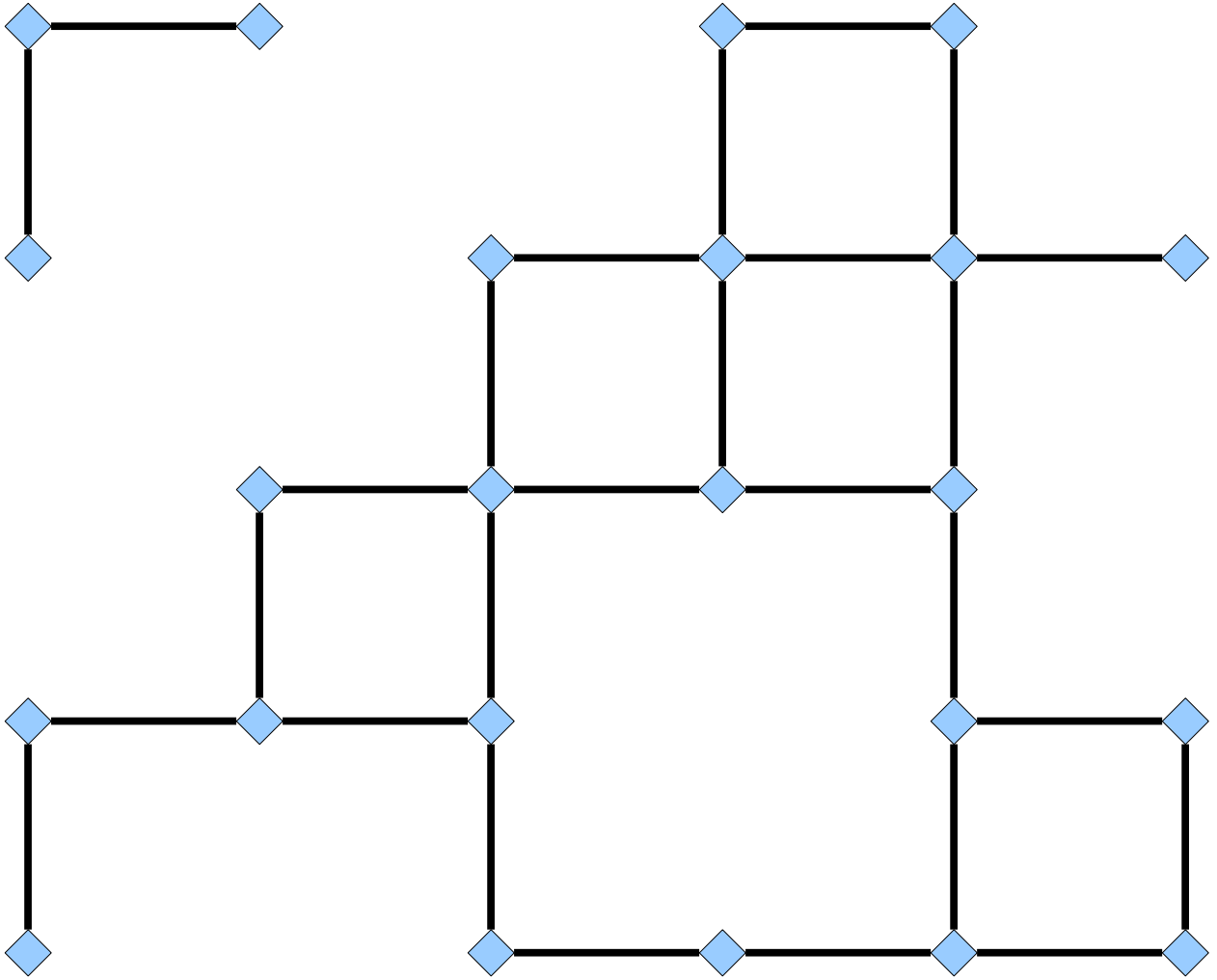
Solving Domino Tiling



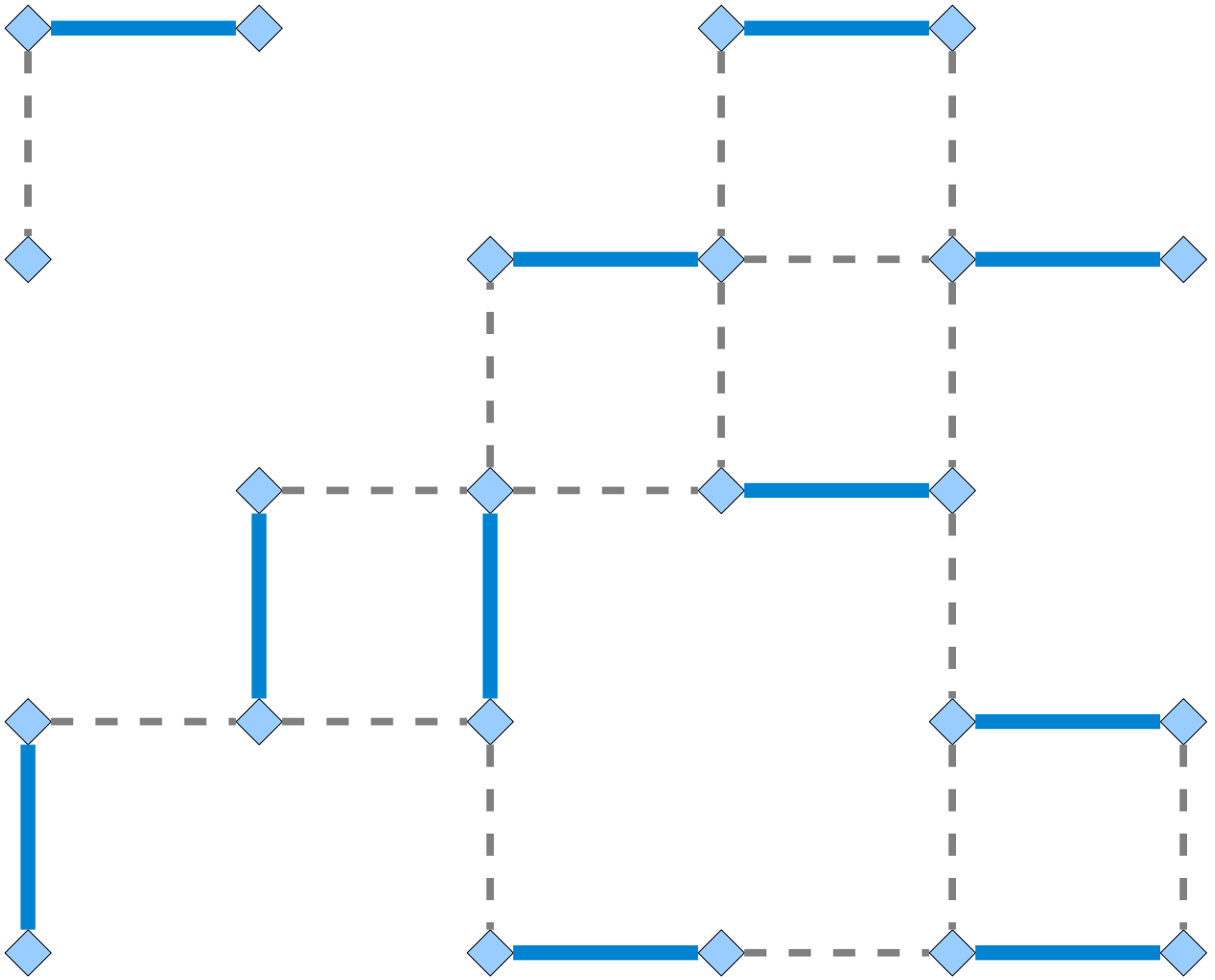
Solving Domino Tiling



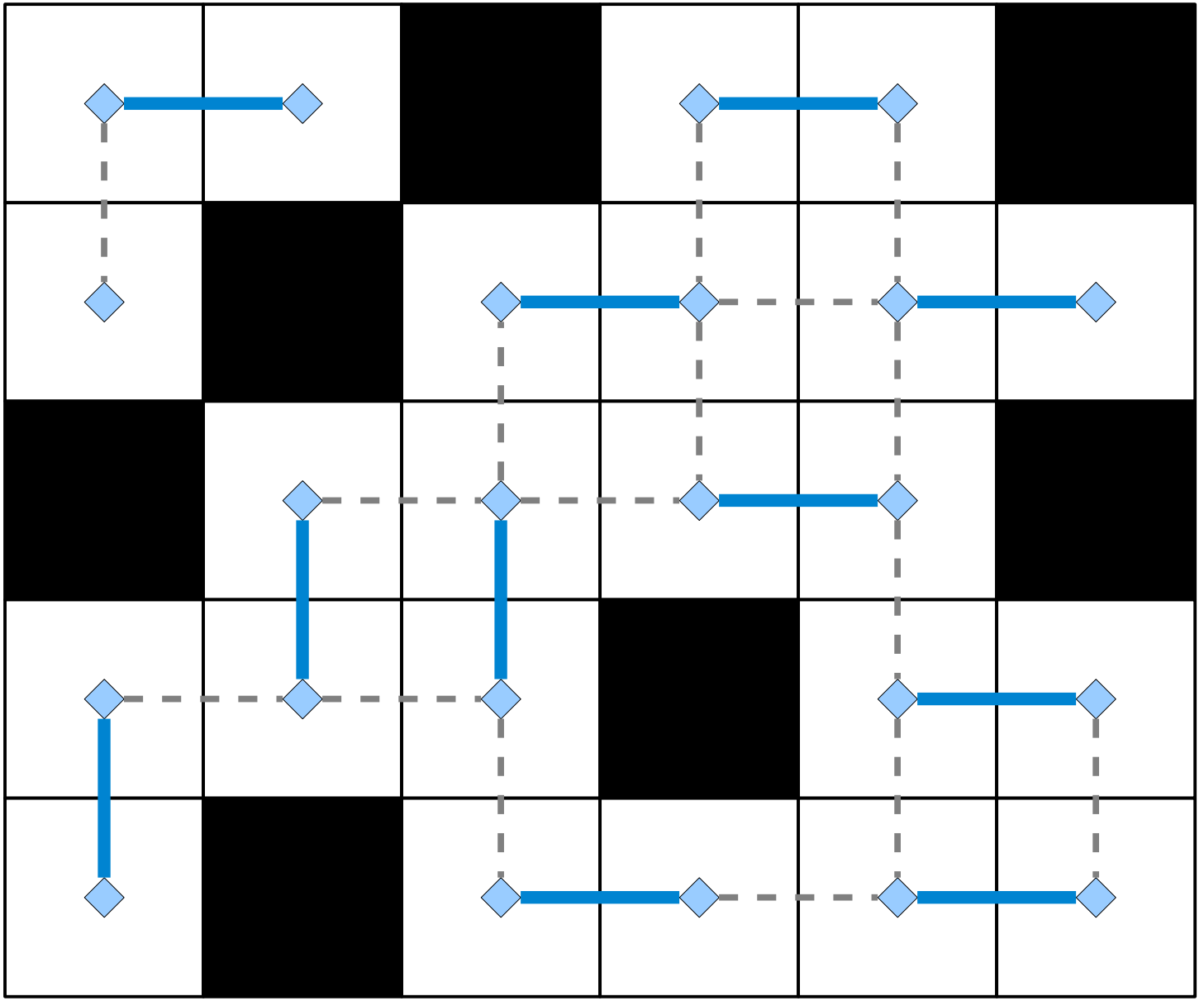
Solving Domino Tiling



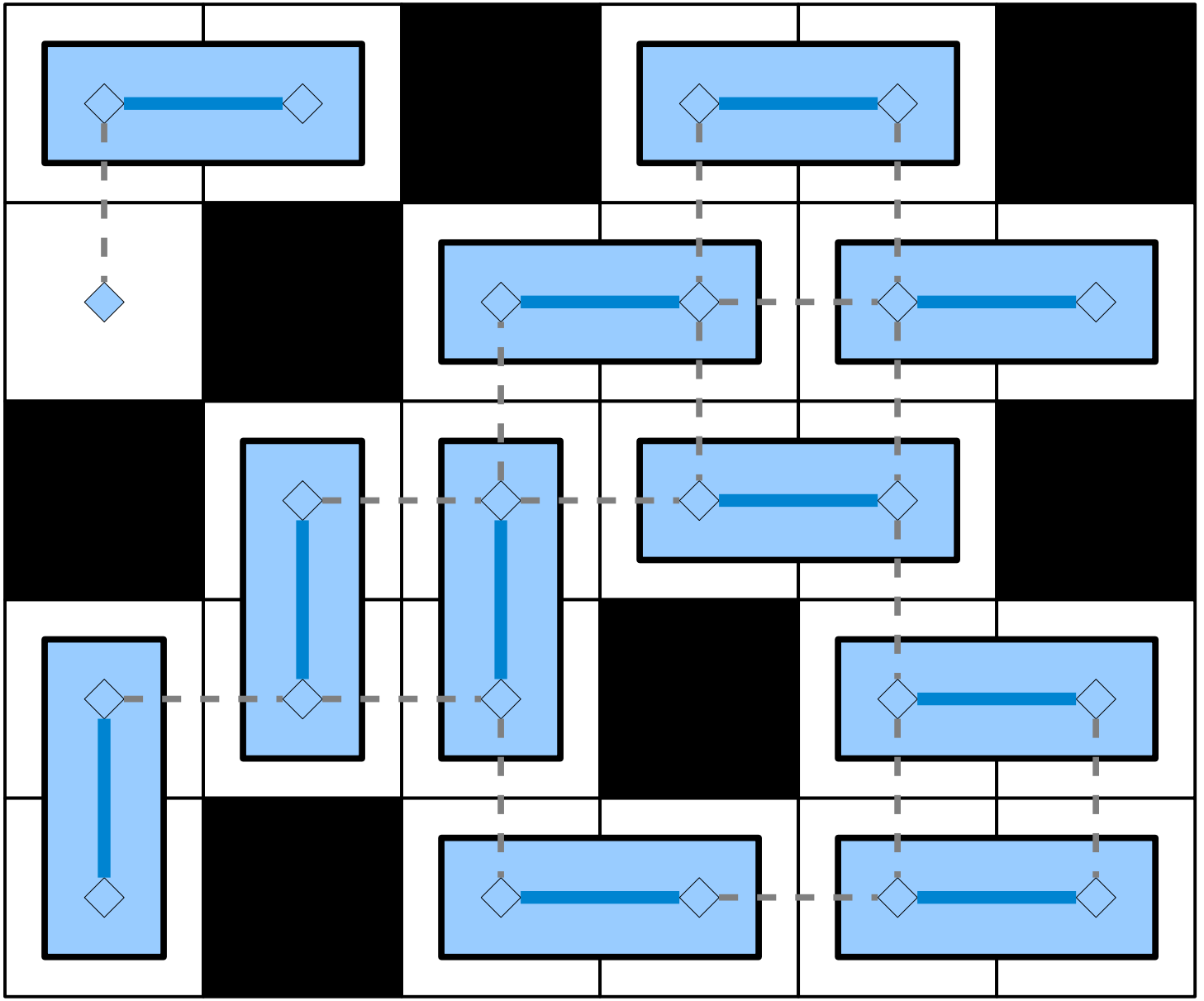
Solving Domino Tiling



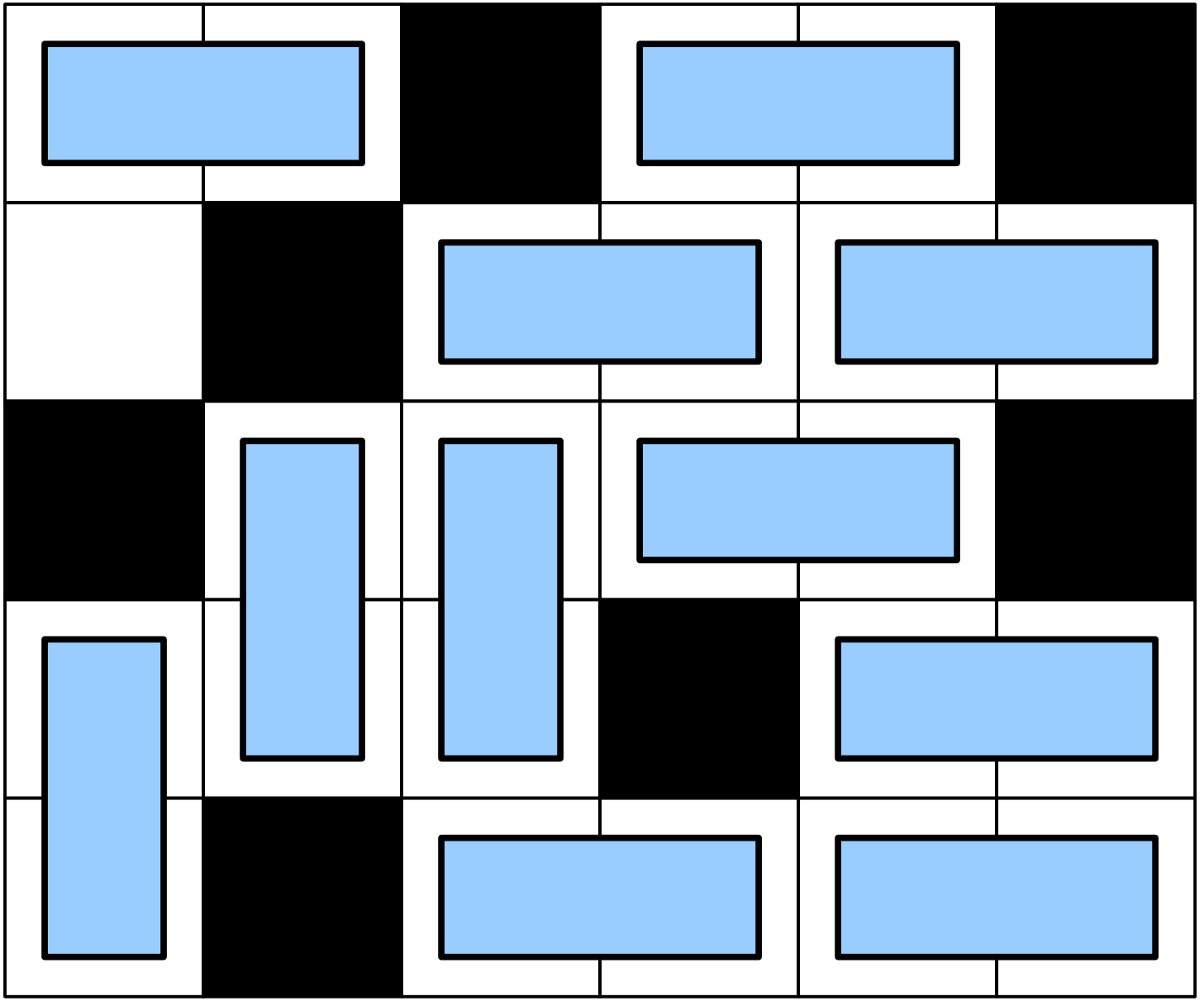
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



In Pseudocode

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

Intuition:

Tiling a grid with dominoes can't be “harder” than solving maximum matching, because if we can solve maximum matching efficiently, we can solve domino tiling efficiently.

Another Example

Reachability

- Consider the following problem:

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

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

Converter Conundrums

Suppose that you want to plug your laptop into a projector.

Your laptop only has a VGA output, but the projector needs HDMI input.

You have a box of connectors that convert various types of input into various types of output (for example, VGA to DVI, DVI to DisplayPort, etc.)

Question: Can you plug your laptop into the projector?

Converter Conundrums

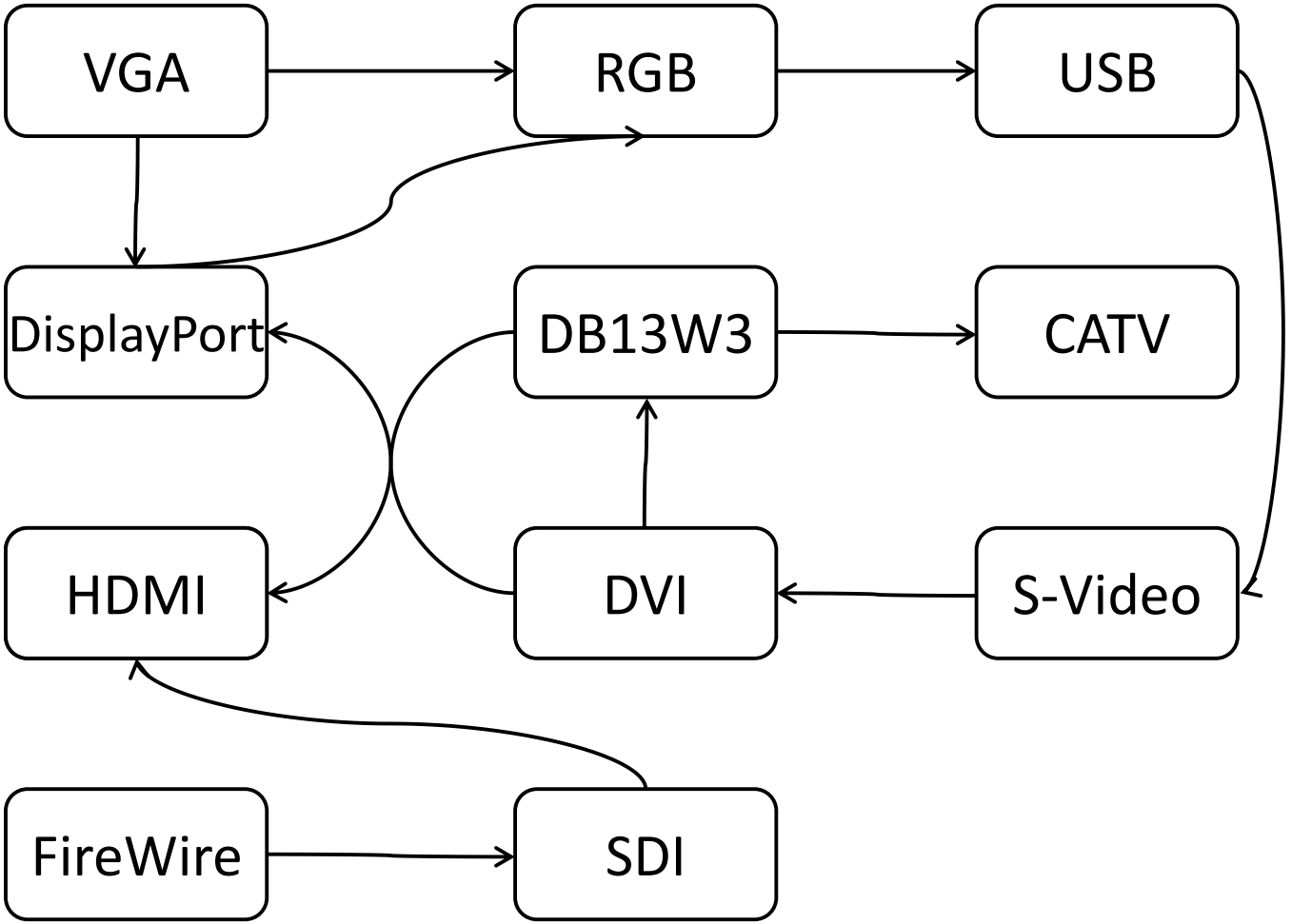
Connectors

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

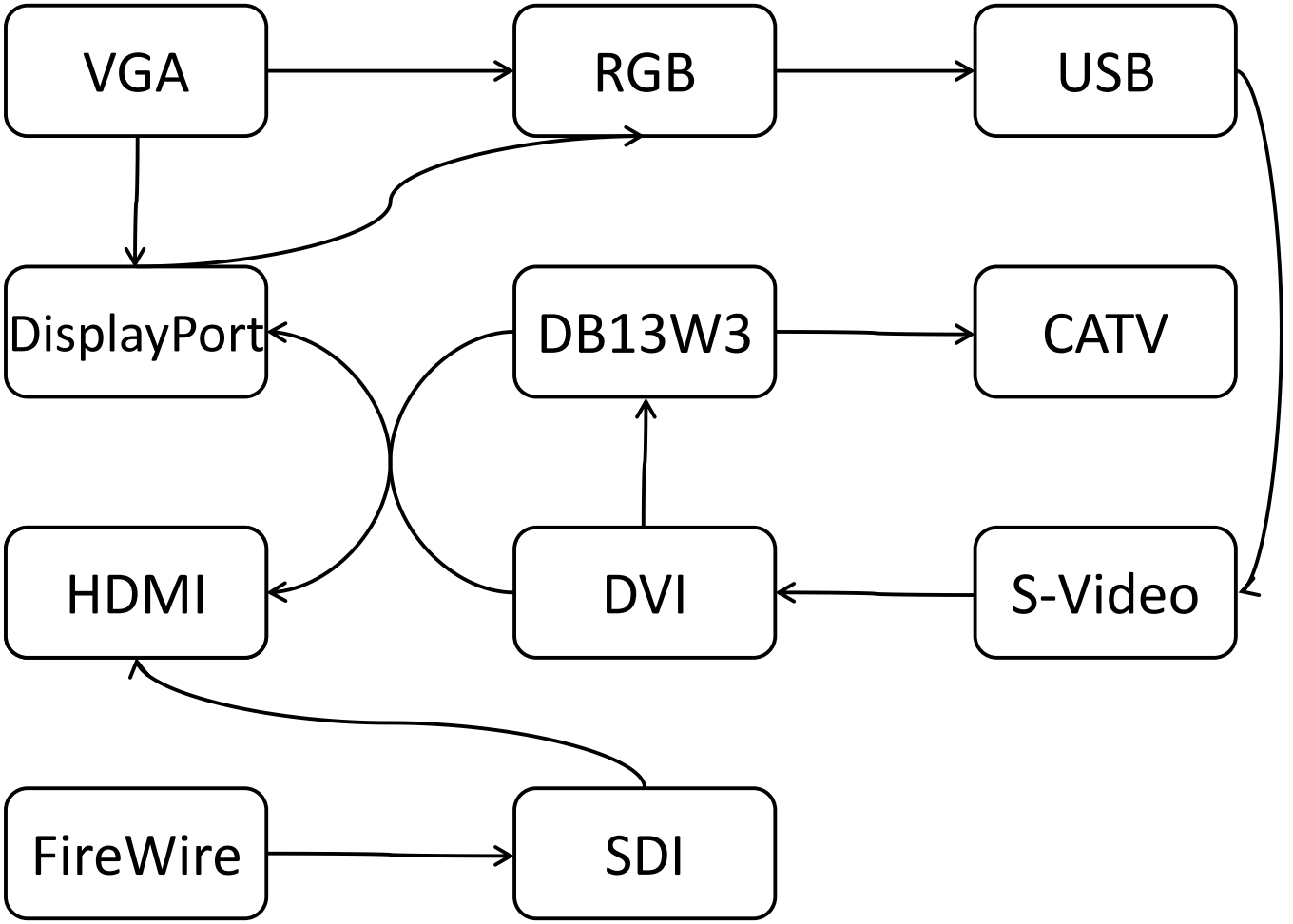
Converter Conundrums

Connectors

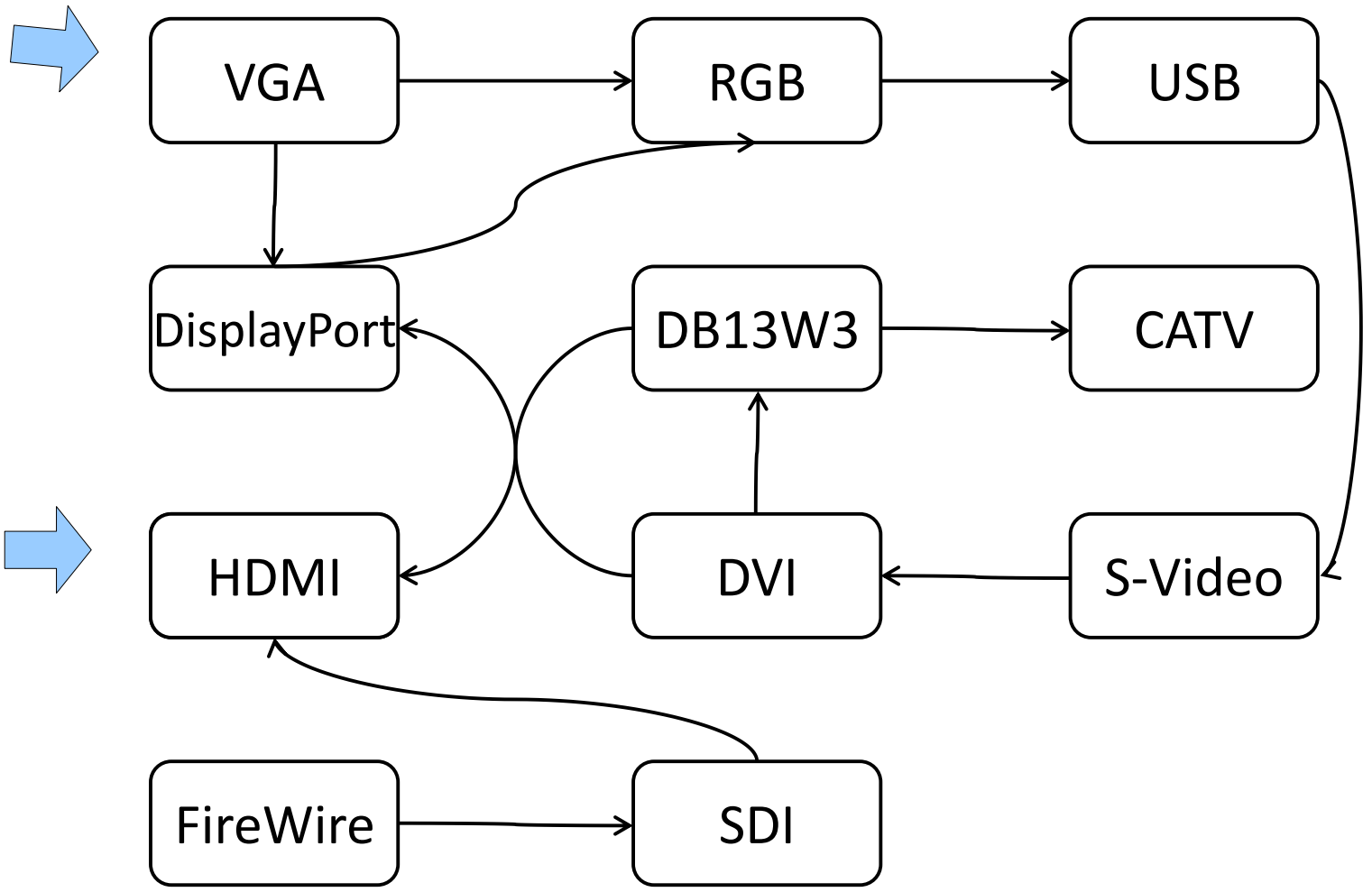
RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



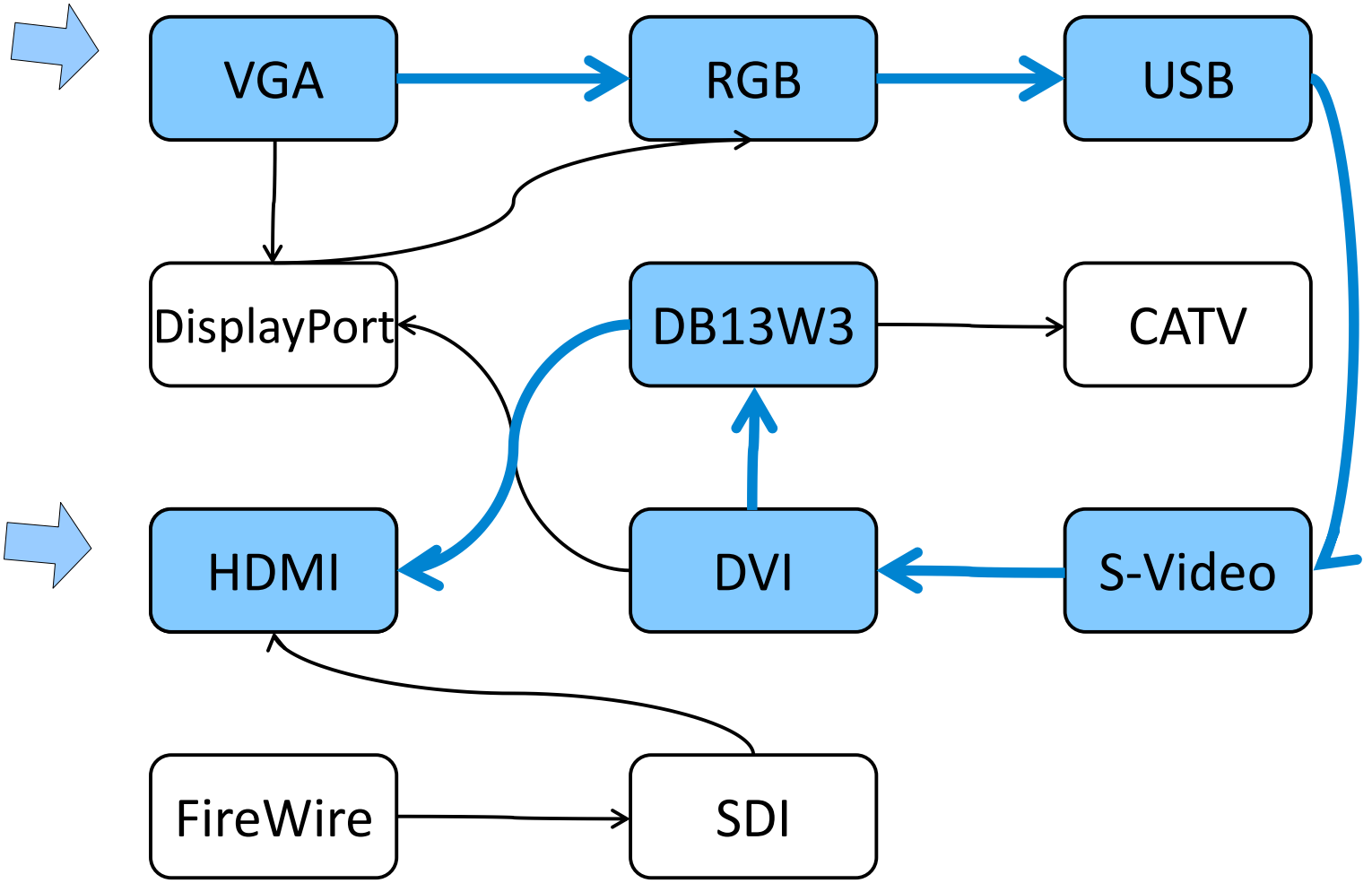
Converter Conundrums



Converter Conundrums



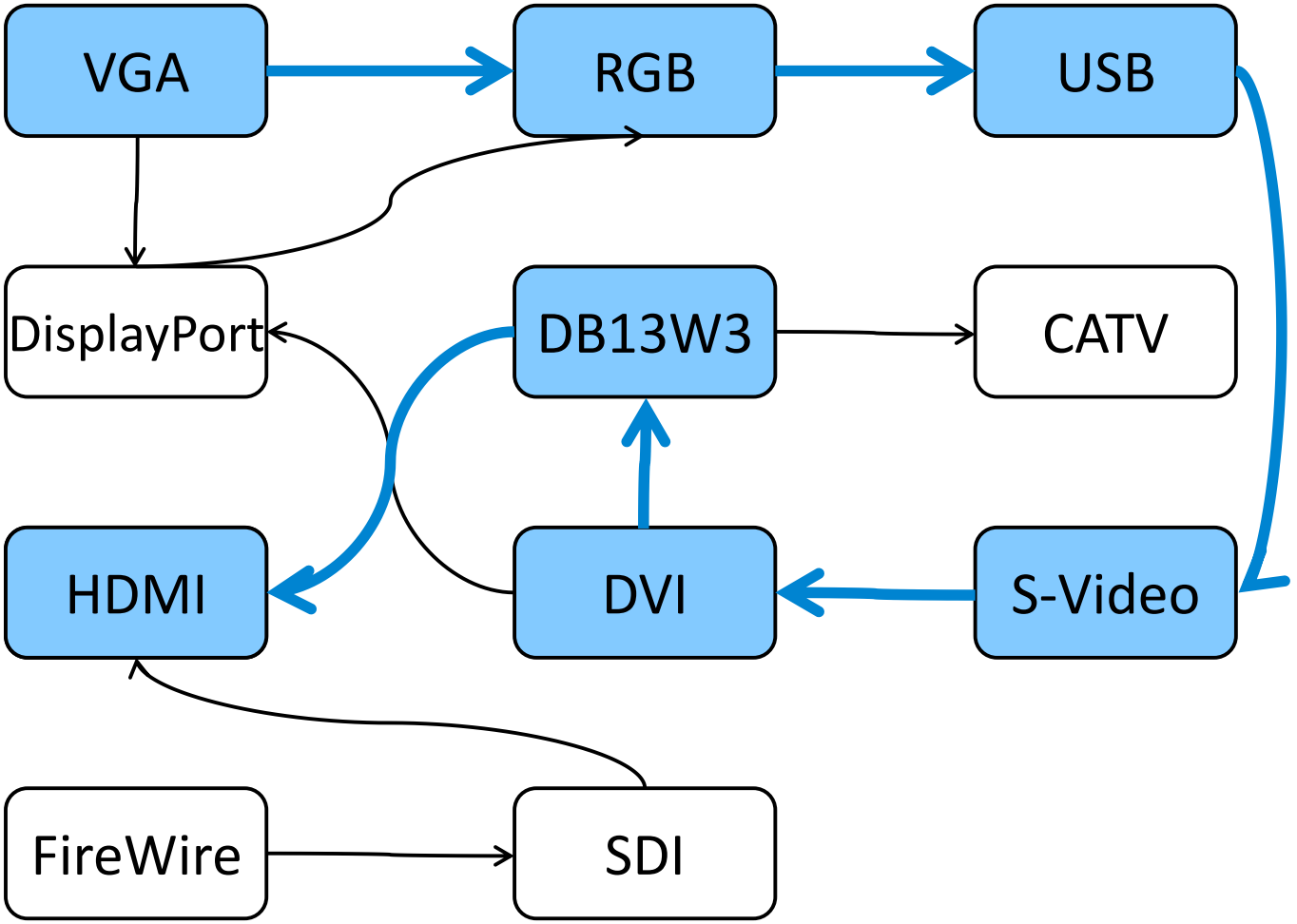
Converter Conundrums



Converter Conundrums

Connectors

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



In Pseudocode

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

Intuition:

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

```
bool solveProblemA(string input) {
    return solveProblemB(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

$$A \leq_{\mathrm{p}} B.$$

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

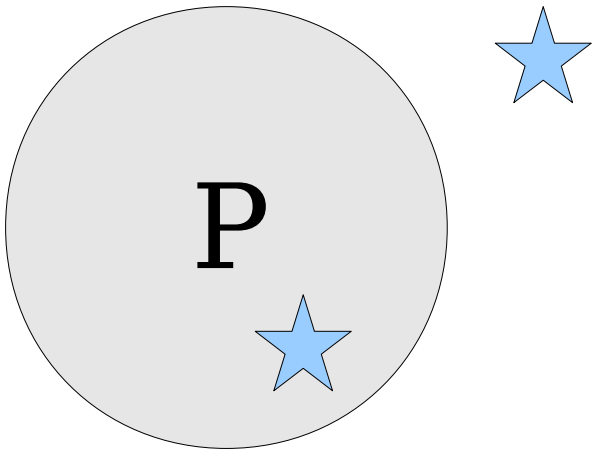
* Assuming that transform runs in polynomial time.

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

- This is a powerful general problem-solving technique. You'll see it a lot in CS161.

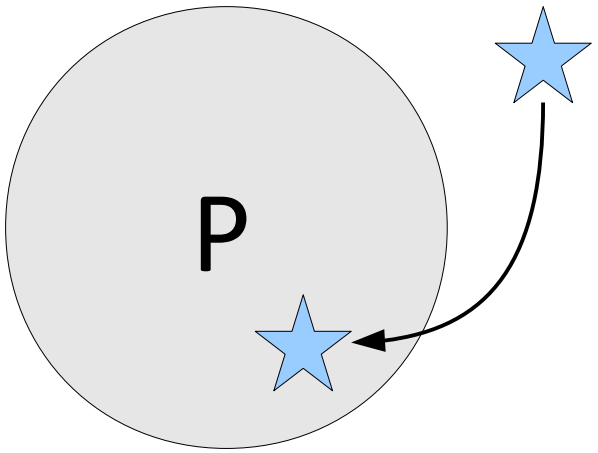
Polynomial-Time Reductions

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



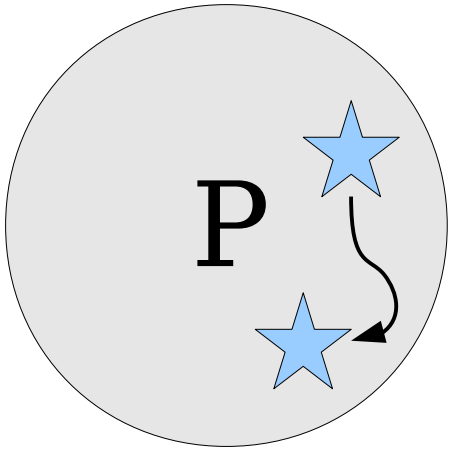
Polynomial-Time Reductions

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



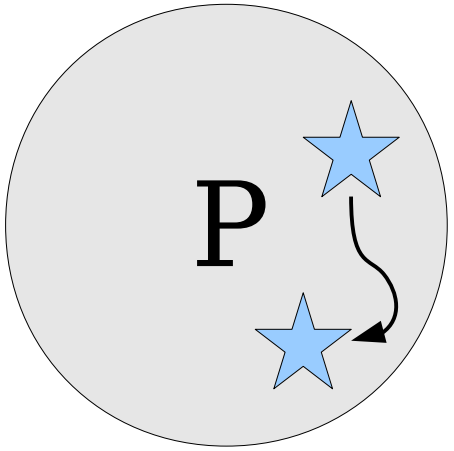
Polynomial-Time Reductions

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



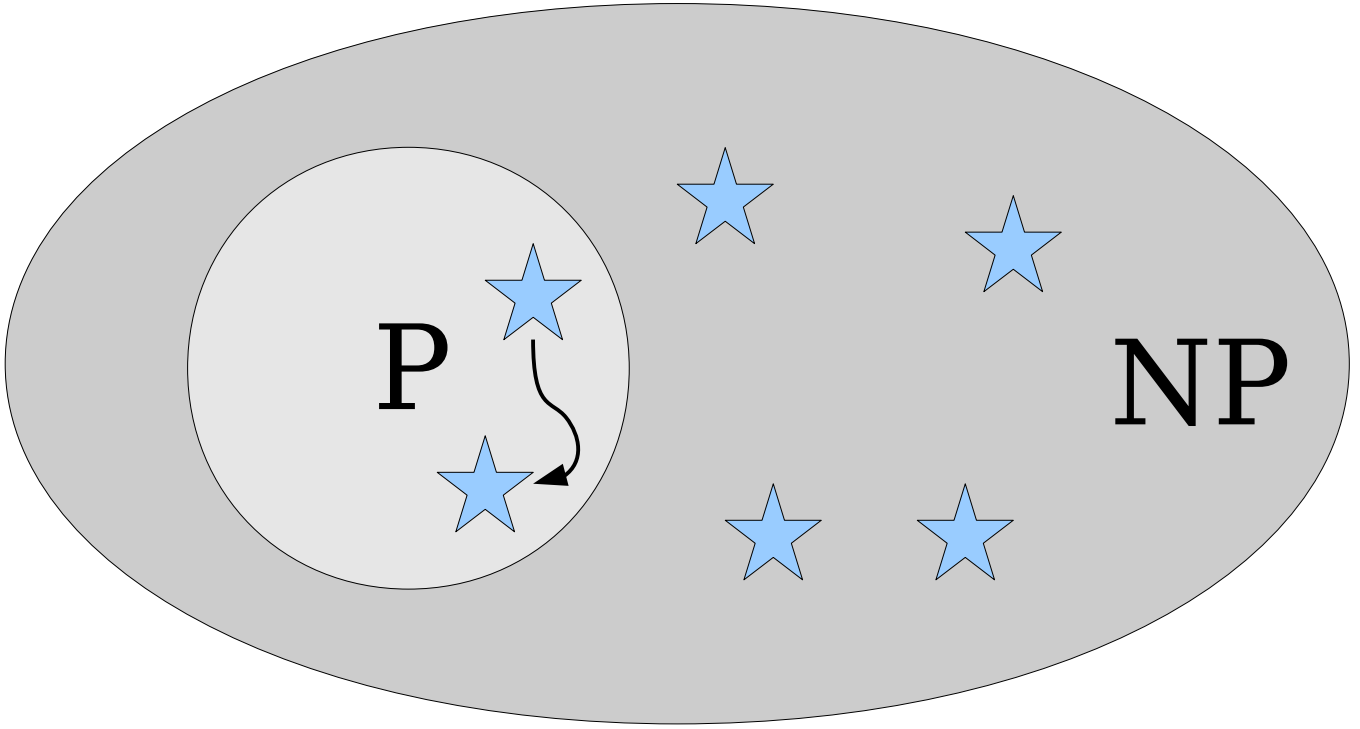
Polynomial-Time Reductions

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



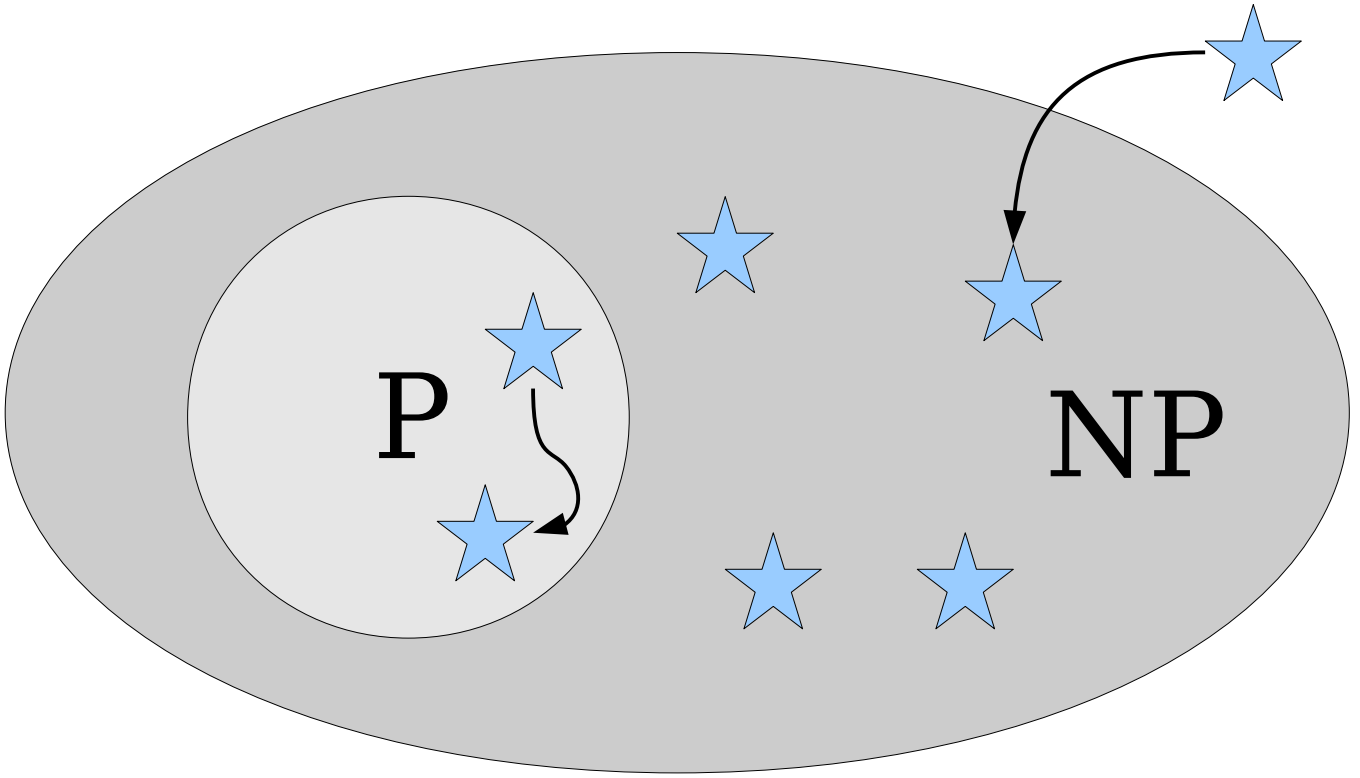
Polynomial-Time Reductions

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



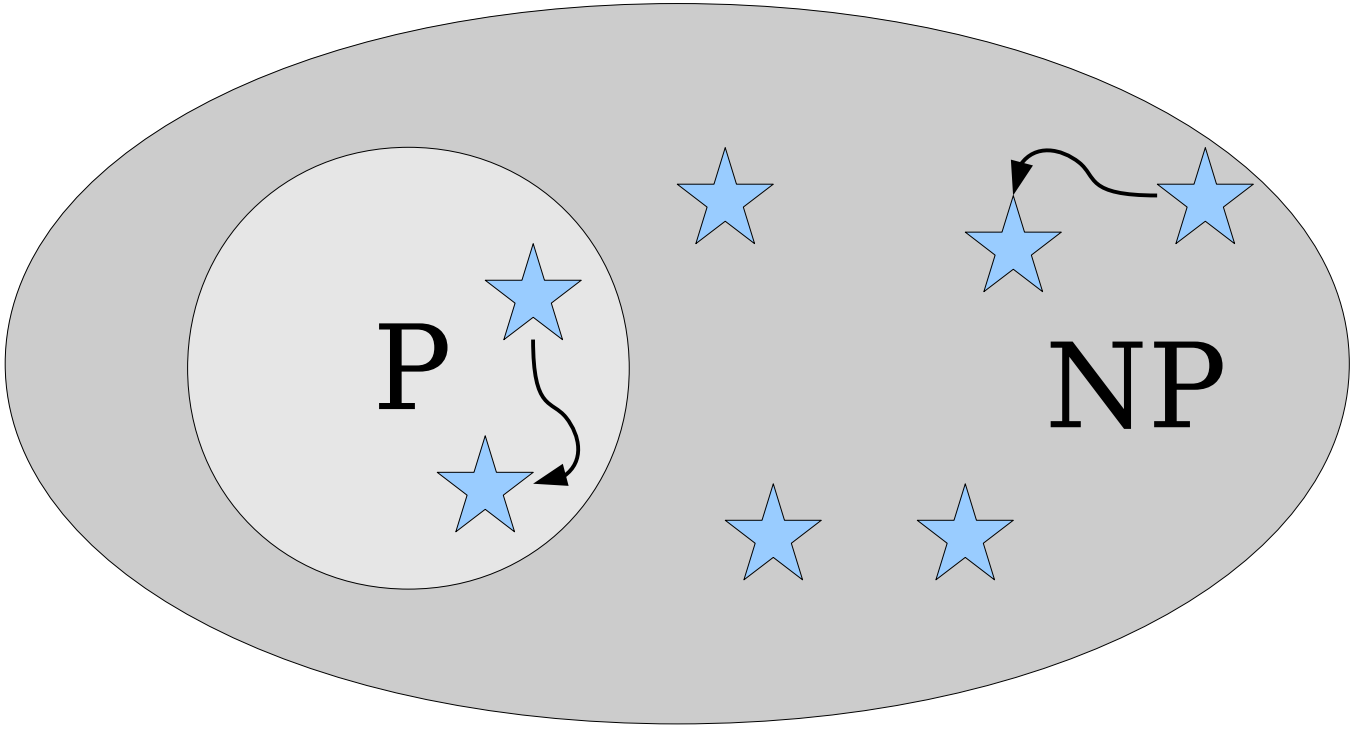
Polynomial-Time Reductions

- 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?

Next Time

NP-Completeness

What are the hardest problems in **NP**?

~~Next Time~~

Right now

NP-Completeness

What are the hardest problems in **NP**?

Complexity Theory

Part Two

Recap from Last Time

The Complexity Class **P**

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

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

Intuitively, **P** contains all decision problems that can be solved efficiently.

This is like class **P**, except with “efficiently” tacked onto the end.

The Complexity Class **NP**

The complexity class **NP** (***nondeterministic polynomial time***) contains all problems that can be verified in polynomial time.

Formally:

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

Intuitively, **NP** is the set of problems where "yes" answers can be checked efficiently.

This is like the class **RE**, but with "efficiently" tacked on to the definition.

The Biggest Unsolved Problem in Theoretical Computer Science:

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

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**?

```
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

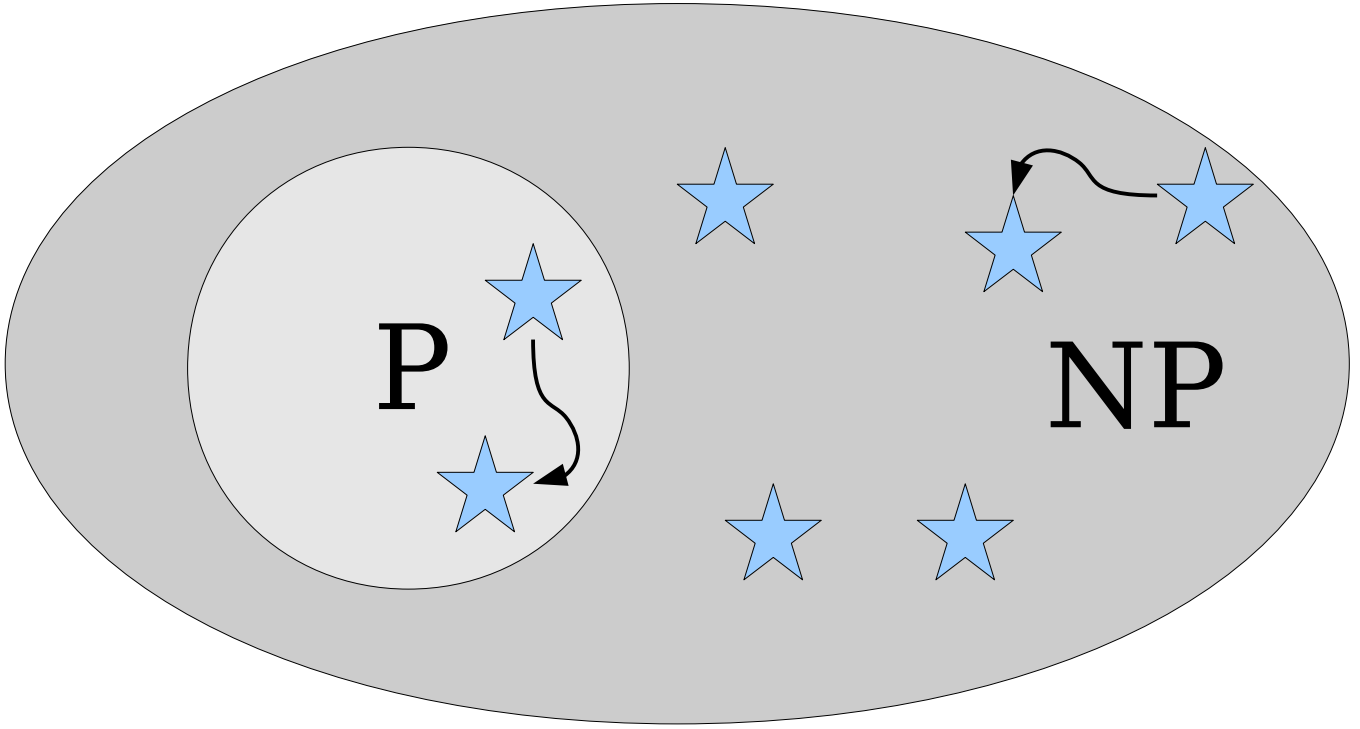
$$A \leq_{\mathrm{p}} B.$$

- 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 $A \in \mathbf{P}$.
- 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?

New Stuff!

Satisfiability

A propositional logic formula φ 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 φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

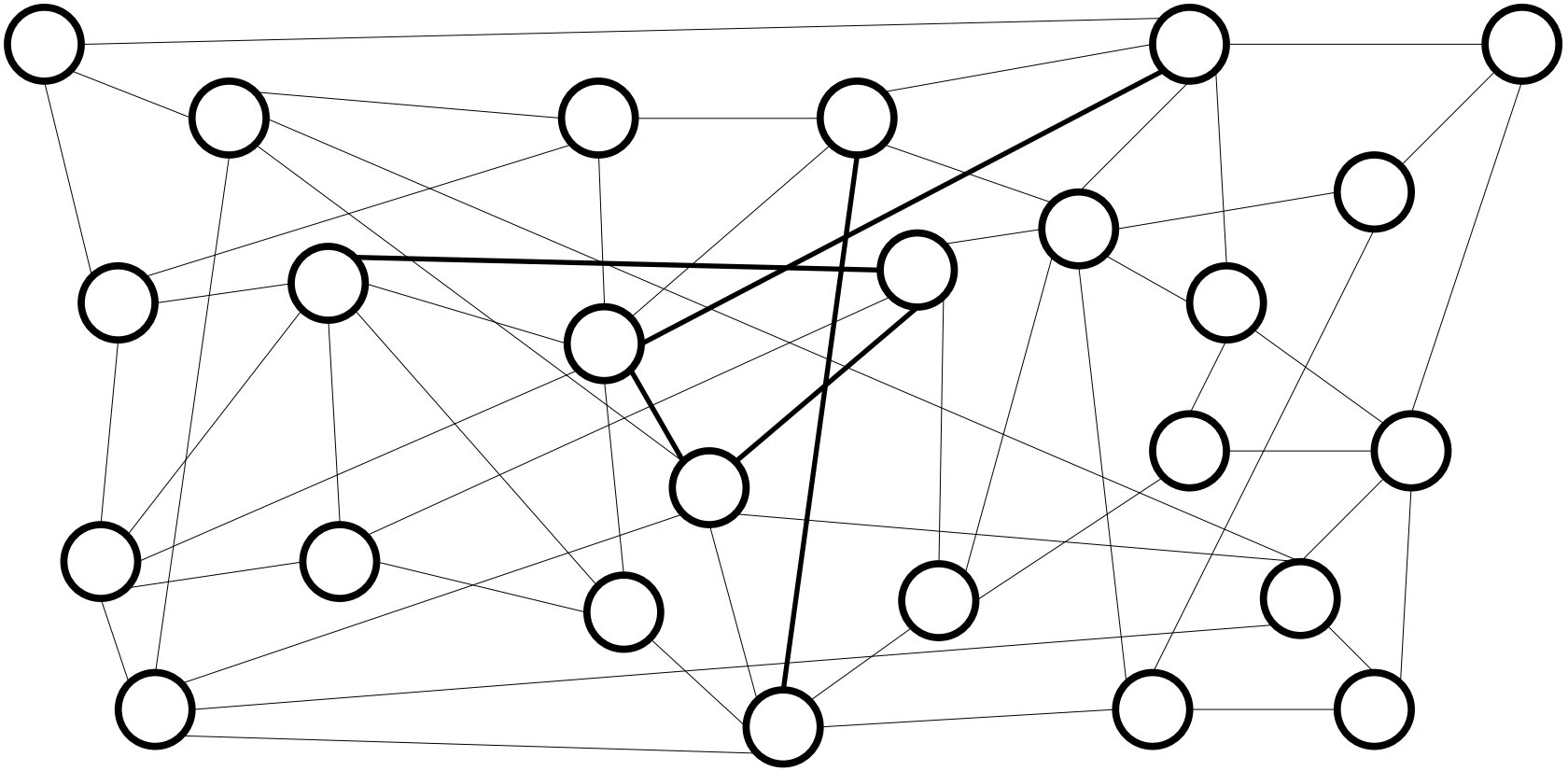
The ***boolean satisfiability problem*** (***SAT***) is the following:

Given a propositional logic formula φ, is φ satisfiable?

Formally:

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

Finding Cliques



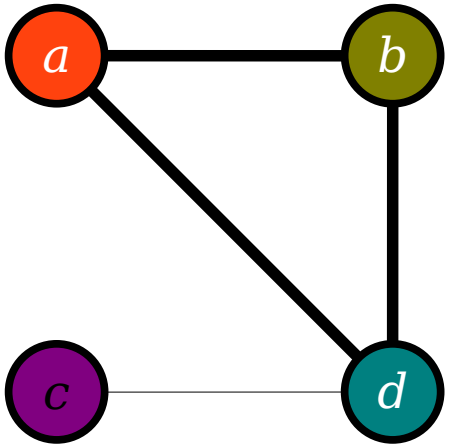
Does this graph contain a k-clique?

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes

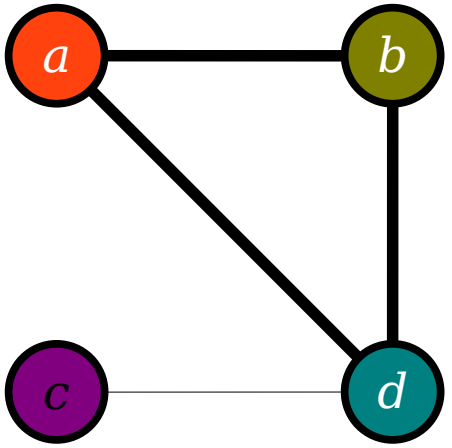


Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



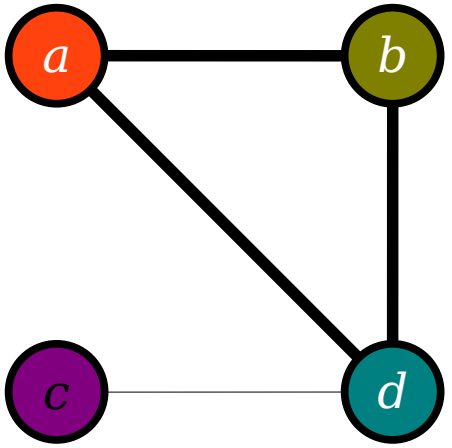
Could we somehow take these rules and encode them as a propositional logic formula?

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

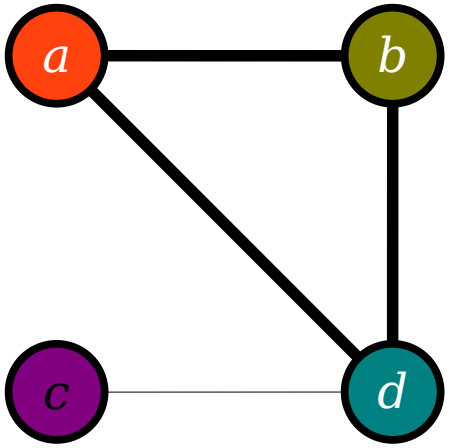
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



Take your graph and define the following propositional variables. The variable c_3 represents choosing node c as the 3rd node of your clique.

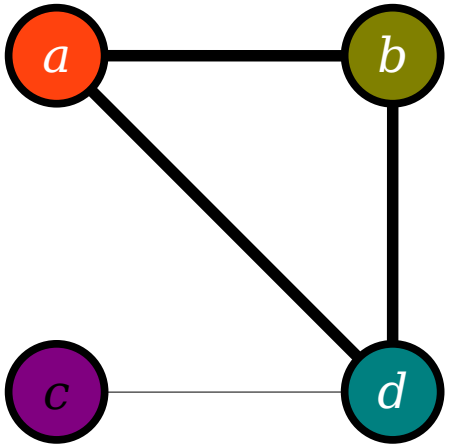
a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

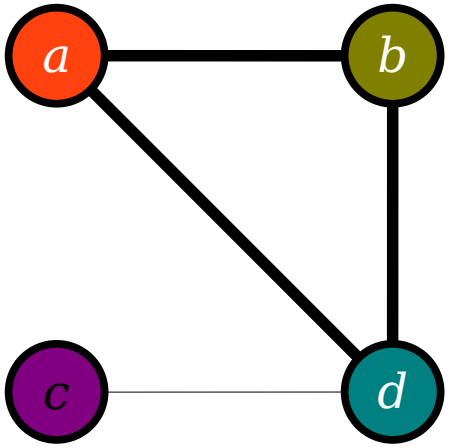
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



Imagine we're looking for a 3-clique. That means we need one variable in each of these groups to be true.

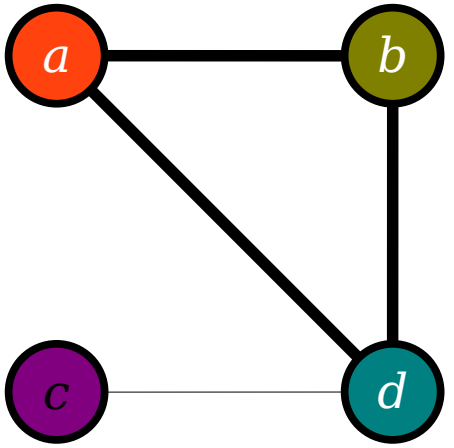
a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



$$(a_1 \vee b_1 \vee c_1 \vee d_1)$$

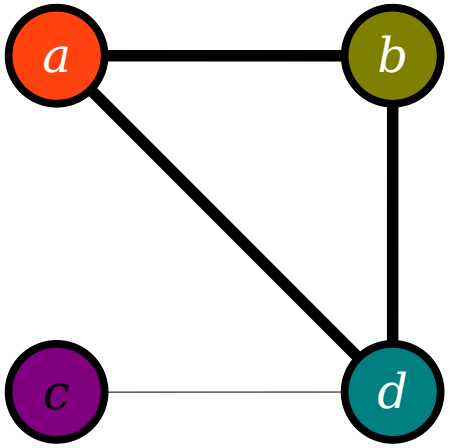
a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



$$(a_1 \vee b_1 \vee c_1 \vee d_1)$$

a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

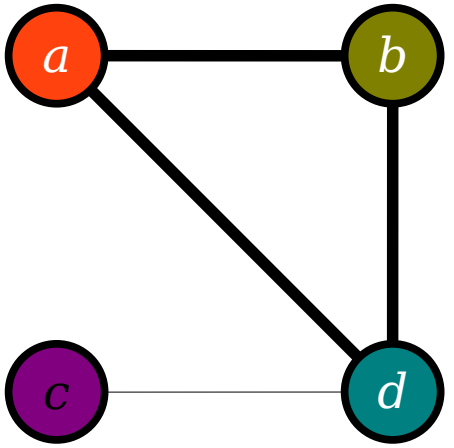
The first node of your clique is either a or b or c or d.

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



$(a_1 \vee b_1 \vee c_1 \vee d_1)$

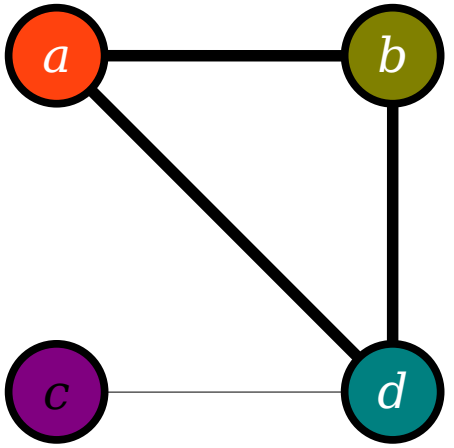
a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



$$(a_1 \vee b_1 \vee c_1 \vee d_1)$$

$$\wedge$$

$$(a_2 \vee b_2 \vee c_2 \vee d_2)$$

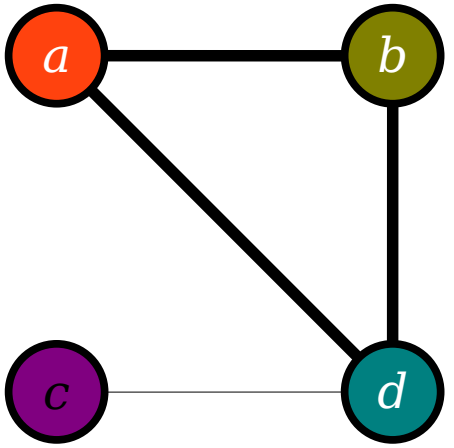
a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



$$(a_1 \vee b_1 \vee c_1 \vee d_1) \wedge (a_2 \vee b_2 \vee c_2 \vee d_2) \wedge (a_3 \vee b_3 \vee c_3 \vee d_3)$$

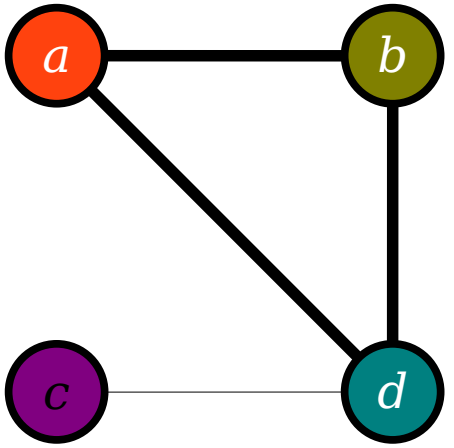
a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



$$(a_1 \vee b_1 \vee c_1 \vee d_1) \wedge (a_2 \vee b_2 \vee c_2 \vee d_2) \wedge (a_3 \vee b_3 \vee c_3 \vee d_3)$$

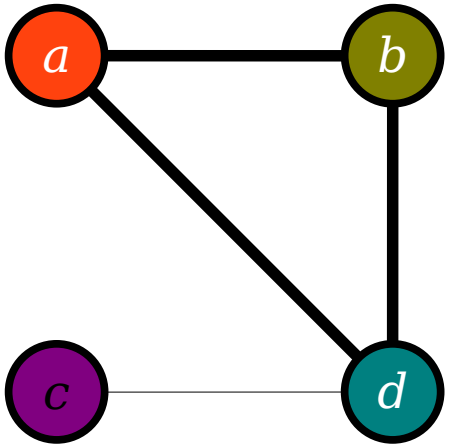
$$\begin{array}{cccc} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{array}$$

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

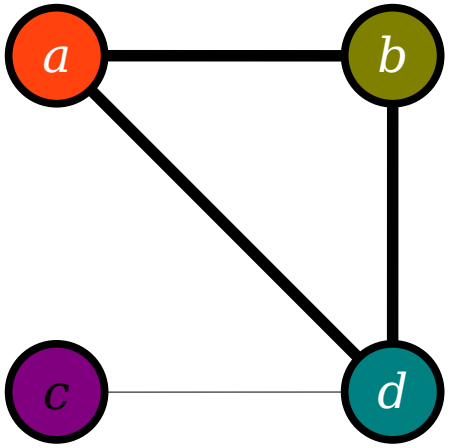
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



We need to ensure we don't pick a pair of nodes that don't have an edge between them. In this graph, the missing edges are $\{a, c\}$ and $\{b, c\}$

a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

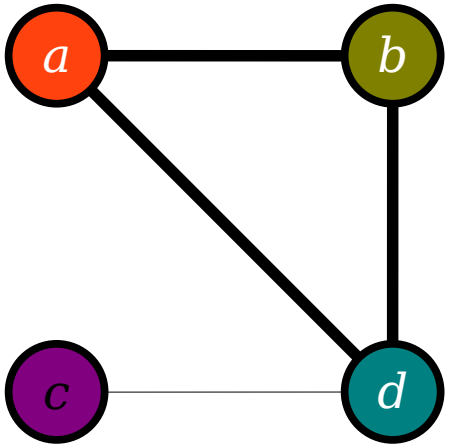
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



for all i, j:

$(\neg a_i \vee \neg c_j)$

a_1	a_2	a_3	a_4
b_1	b_2	b_3	b_4
c_1	c_2	c_3	c_4
d_1	d_2	d_3	d_4

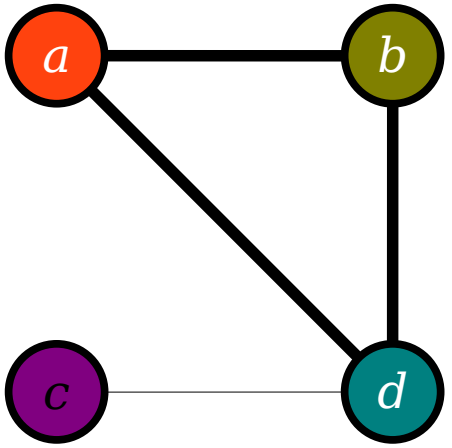
We can't choose both a and c because there's no edge between them.

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



for all i, j:
$(\neg a_i \vee \neg c_j)$

for all i, j:
$(\neg b_i \vee \neg c_j)$

a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

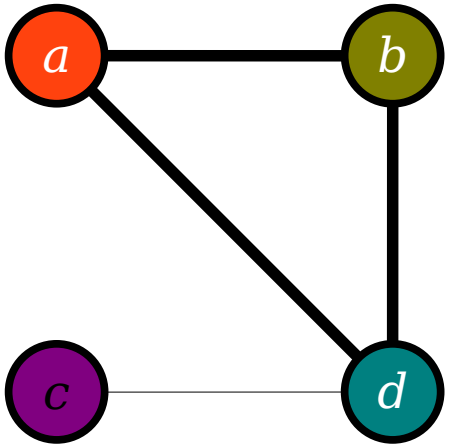
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

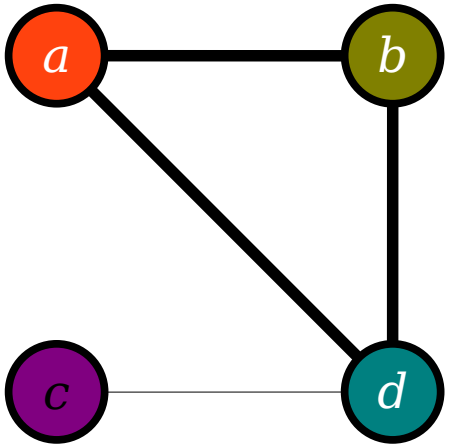
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



Altogether, finding an assignment of true false values to these variables that satisfies these constraints would amount to finding a clique of desired size in our graph.

a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

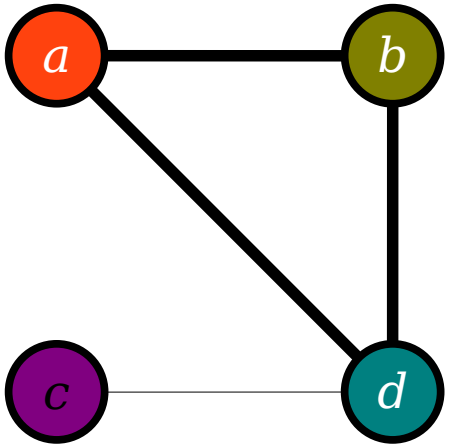
d_1 d_2 d_3 d_4

Finding Cliques

What is a k-clique?

A set of k nodes

Such that there's an edge between every pair of nodes



a_1 a_2 a_3 a_4

b_1 b_2 b_3 b_4

c_1 c_2 c_3 c_4

d_1 d_2 d_3 d_4

One example that works is to assign a_1, b_2, and d_3 to be true and all other variables to be false.

Intuition:

Finding a k-clique can't be any harder than solving *SAT*, because we can take any graph and encode it as a propositional logic formula where finding a satisfying assignment corresponds to finding a k-clique.

Solving Sudoku

		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 puzzle have a solution?

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - Subject to some existing constraints (numbers that have been filled in already)

		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		

Solving Sudoku

What is a Sudoku solution?

- **An assignment of numbers 1 through 9 to a 9×9 grid**
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - Subject to some existing constraints (numbers that have been filled in already)

		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		

Take your puzzle and let the variable $x_{i,j,k}$ represent filling in cell (i, j) with value k.

Solving Sudoku

What is a Sudoku solution?

- **An assignment of numbers 1 through 9 to a 9×9 grid**
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - Subject to some existing constraints (numbers that have been filled in already)

		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		

for all i, j:

$x_{i,j,1} \vee \ldots \vee x_{i,j,9}$

All cells (i, j) should have some value between 1 and 9.

Solving Sudoku

What is a Sudoku solution?

- **An assignment of numbers 1 through 9 to a 9×9 grid**
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - Subject to some existing constraints (numbers that have been filled in already)

		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		

for all i, j:

$$x_{i,j,1} \vee \ldots \vee x_{i,j,9}$$

$$\wedge$$

for all $k \neq l$:

$$(\neg x_{i,j,k} \vee \neg x_{i,j,l})$$

And each cell should only be assigned one value.

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
- **Such that each number appears exactly once in each row, column, and 3×3 square**
- Subject to some existing constraints (numbers that have been filled in already)

		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		

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
- **Such that each number appears exactly once in each row, column, and 3×3 square**
- Subject to some existing constraints (numbers that have been filled in already)

		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		

for all k in [1, 9]:

$(x_{1,1,k} \vee \ldots \vee x_{1,9,k})$

In the first row, every value k has to be assigned to one of the cells.

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
- **Such that each number appears exactly once in each row, column, and 3×3 square**
- Subject to some existing constraints (numbers that have been filled in already)

		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		

Add similar constraints for the other rows, columns, and 3x3 squares!

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - **Subject to some existing constraints (numbers that have been filled in already)**

		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		

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
- Such that each number appears exactly once in each row, column, and 3×3 square
- **Subject to some existing constraints (numbers that have been filled in already)**

		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		

$(x_{1,3,7}$

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - **Subject to some existing constraints (numbers that have been filled in already)**

		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		

$(x_{1,3,7} \wedge x_{1,5,6}$

Solving Sudoku

What is a Sudoku solution?

- An assignment of numbers 1 through 9 to a 9×9 grid
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - **Subject to some existing constraints (numbers that have been filled in already)**

		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		

$$(X_{1,3,7} \wedge X_{1,5,6} \wedge X_{1,6,1} \wedge \ldots)$$

Solving Sudoku

What is a Sudoku solution?

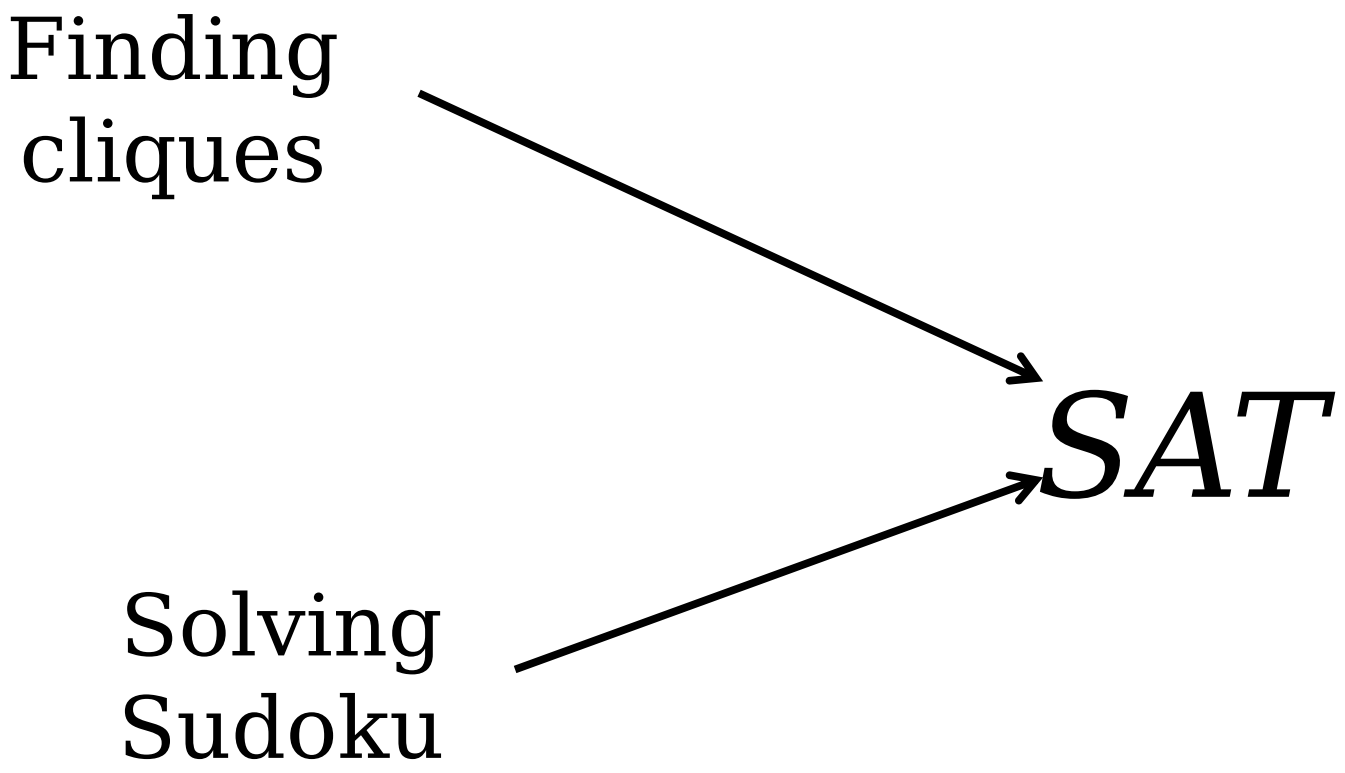
- An assignment of numbers 1 through 9 to a 9×9 grid
  - Such that each number appears exactly once in each row, column, and 3×3 square
  - Subject to some existing constraints (numbers that have been filled in already)

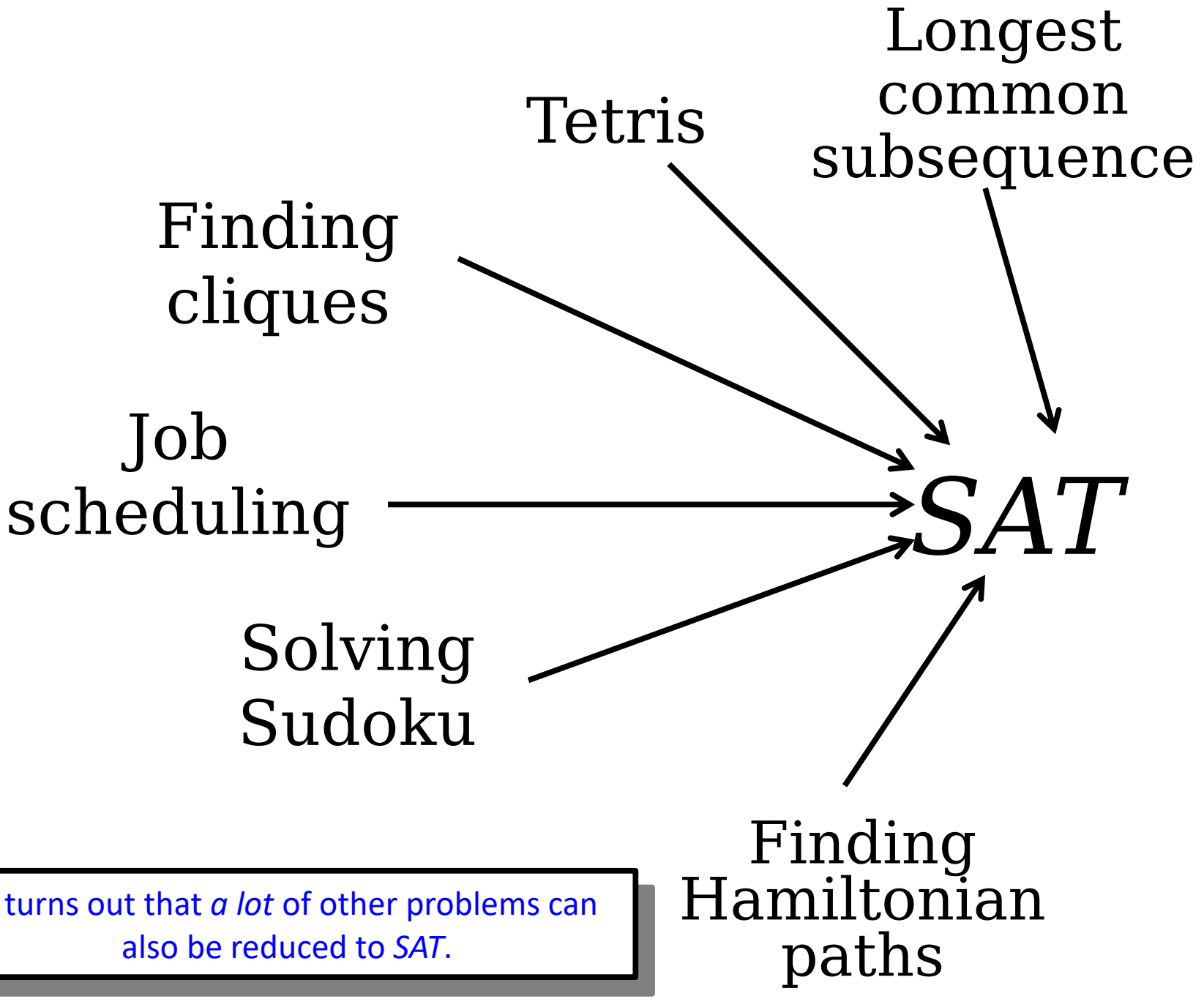
		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

Altogether, finding an assignment of true false values to these variables that satisfies these constraints would amount to finding a solution to our Sudoku puzzle.

Intuition:

Solving Sudoku can’t be any harder than solving *SAT*, because we can take any Sudoku puzzle and encode it as a propositional logic formula where finding a satisfying assignment corresponds to finding a puzzle solution.


Finding
cliques
SAT
Solving
Sudoku



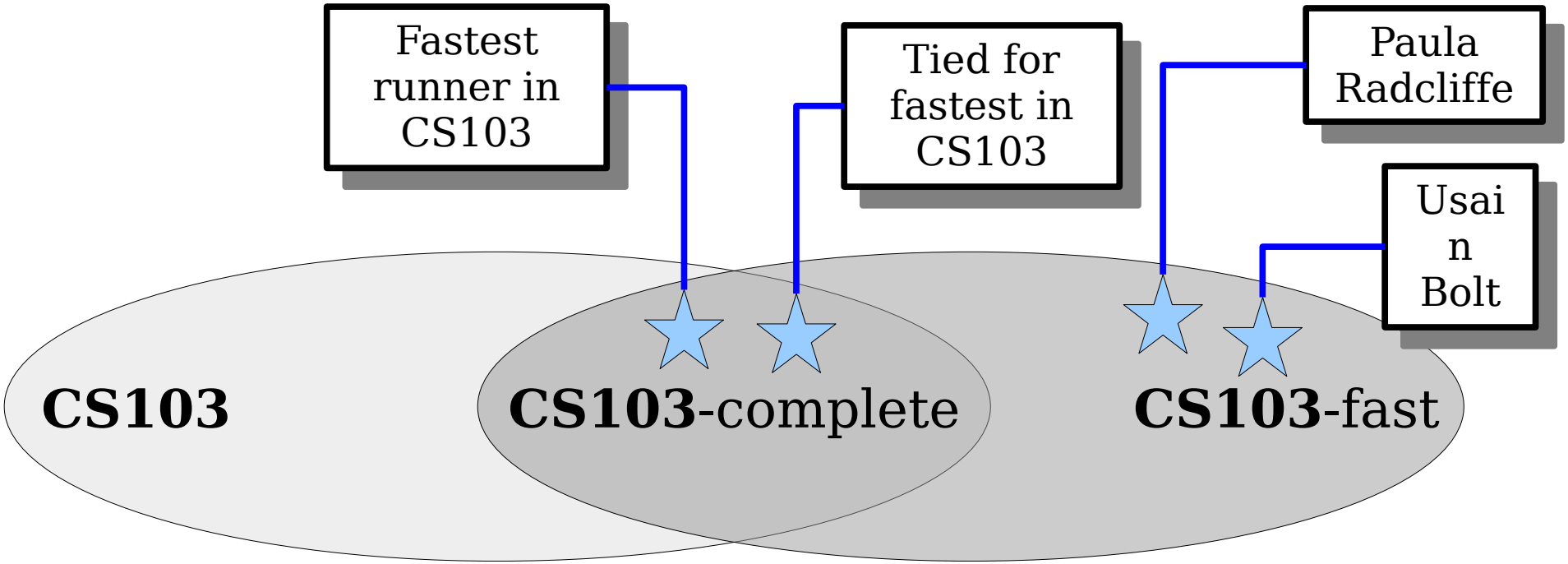
It turns out that *a lot* of other problems can also be reduced to *SAT*.

Key Observations:

(1) *SAT* is ***versatile*** – being able to solve *SAT* allows us to solve many other problems.

(2) The fact that lots of problems reduce to *SAT* suggests we can gauge the difficulty of these problems by looking at the difficulty of *SAT*.

An Analogy: Running Really Fast



For people A and B, we say $A \leq_r B$ if A's top running speed is at most B's top speed.

(Intuitively: B can run at least as fast as A.)

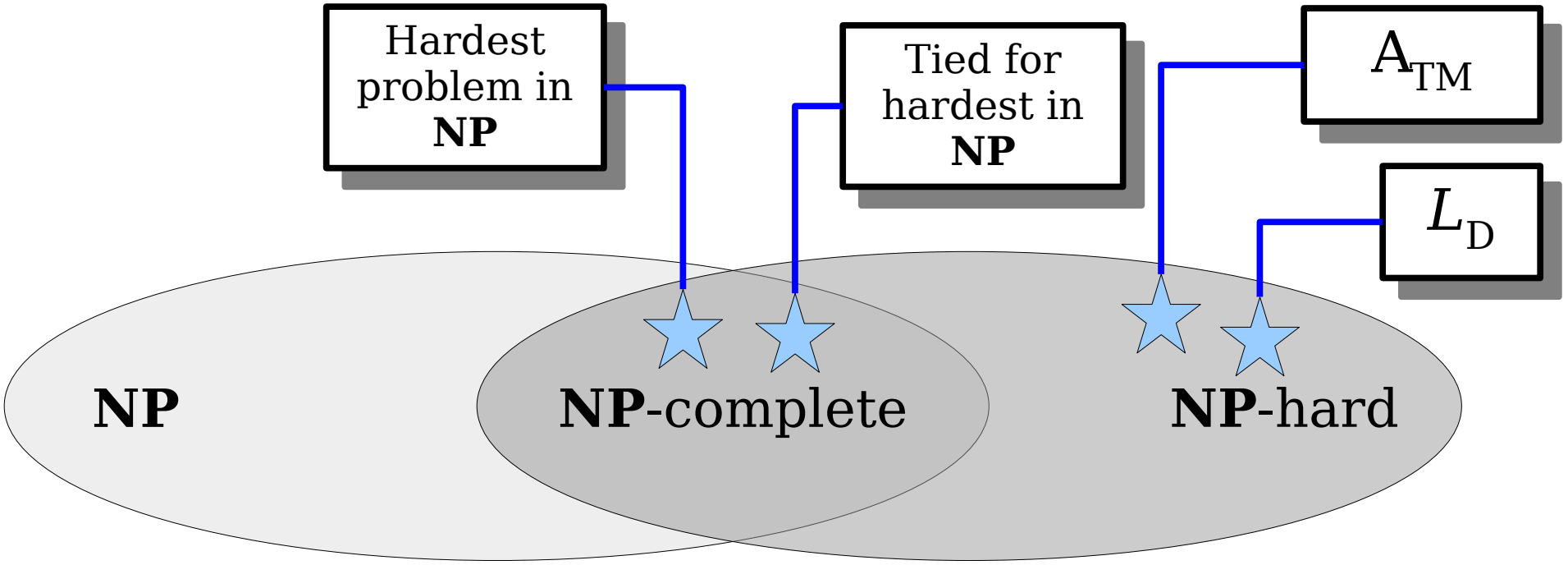
We say that person P is ***CS103-fast*** if

$\forall A \in \textbf{CS103}.\ A \leq_r P.$

(How fast are you if you're CS103-fast?)

We say that person P is ***CS103-complete*** if $P \in \textbf{CS103}$ and P is **CS103**-fast.

(How fast are you if you're CS103-complete?)



For languages A and B, we say $A \leq_p B$ if A reduces to B in polynomial time.

(Intuitively: B is at least as hard as A.)

We say that a language L is ***NP-hard*** if

$\forall A \in \mathbf{NP}.\ A \leq_p L.$

(How hard is a problem that's NP-hard?)

We say that a language L is ***NP-complete*** if

$L \in \mathbf{NP}$ and L is **NP**-hard.

(How hard is a problem that's NP-complete?)

Intuition: The **NP**-complete problems are the hardest problems in **NP**.

If we can determine how hard those problems are, it would tell us a lot about the **P** $\stackrel{?}{=}$ **NP** question.

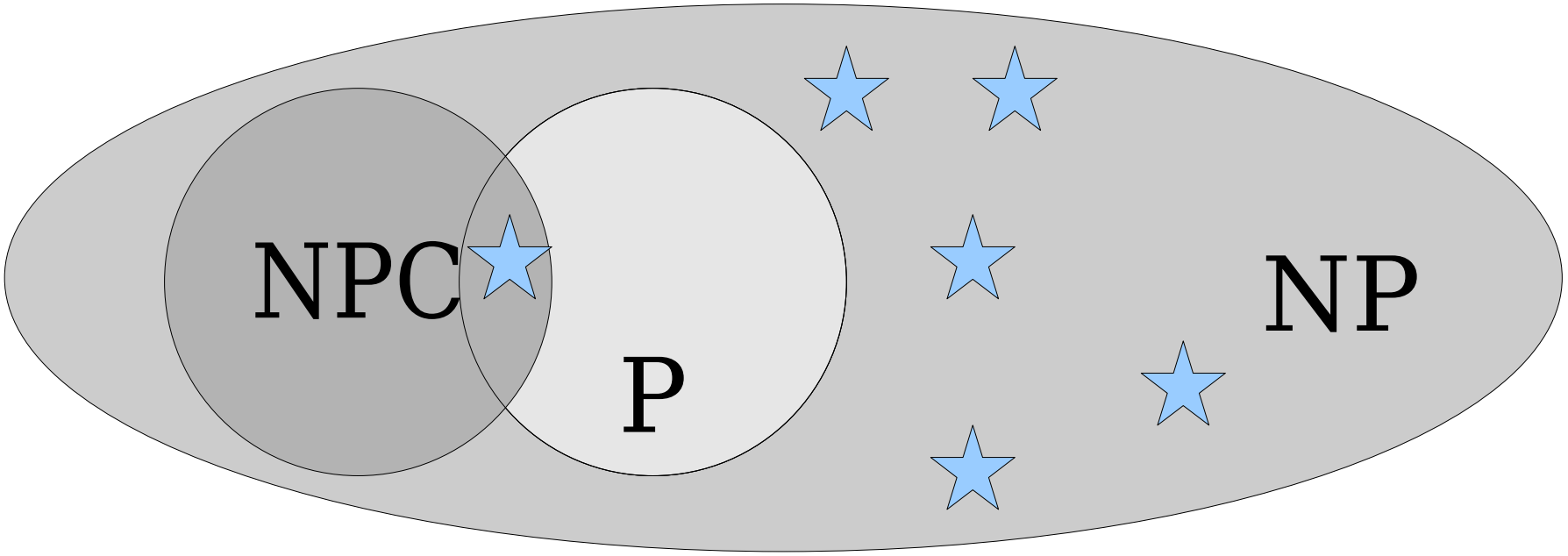
The Tantalizing Truth

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

Intuition: This means the hardest problems in **NP** aren't actually that hard. We can solve them in polynomial time. So that means we can solve all problems in **NP** in polynomial time.

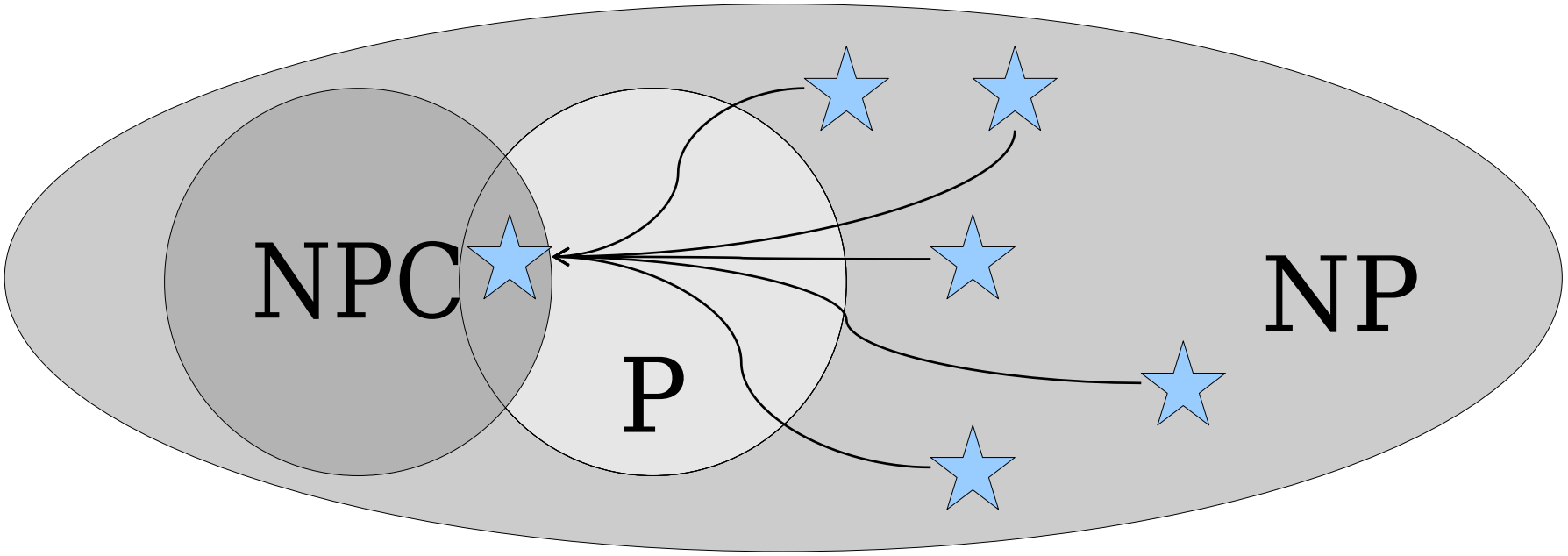
The Tantalizing Truth

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



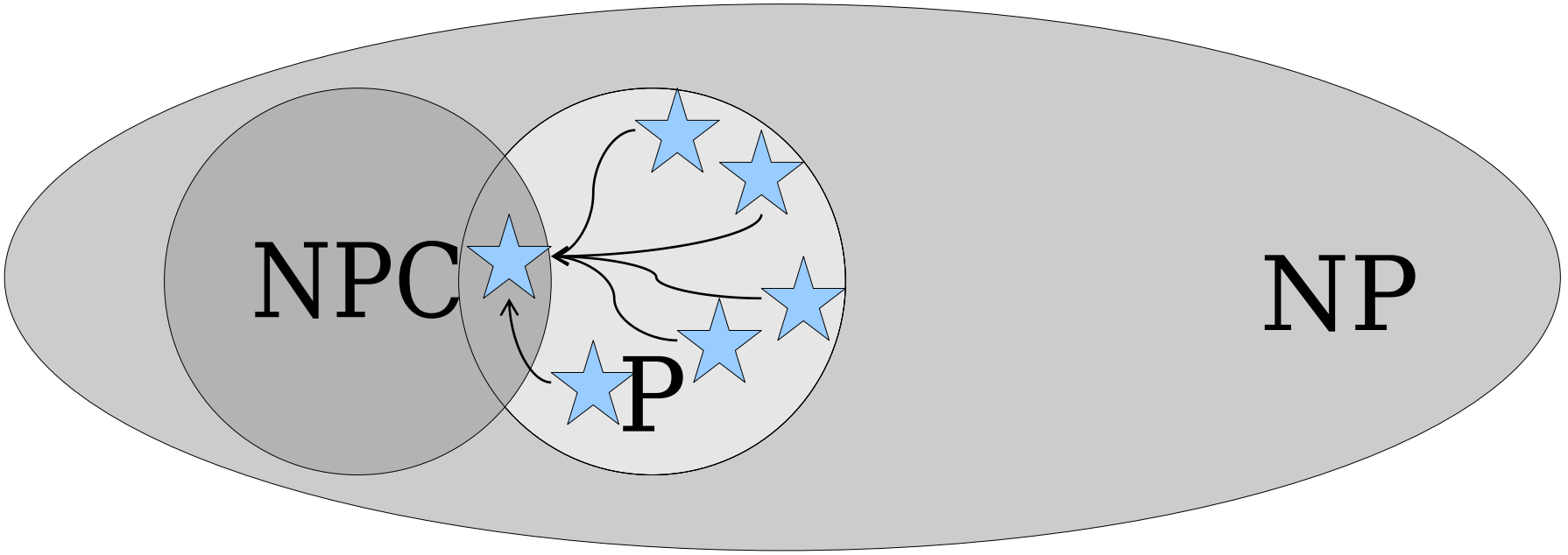
The Tantalizing Truth

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



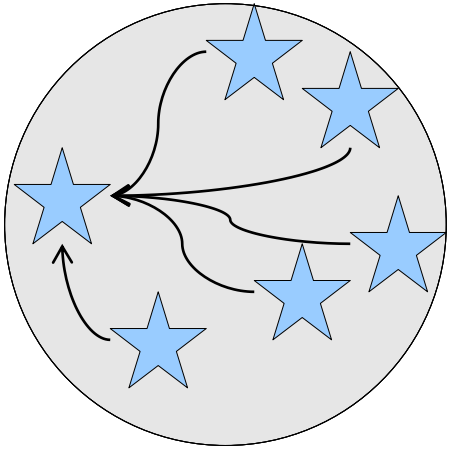
The Tantalizing Truth

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



The Tantalizing Truth

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

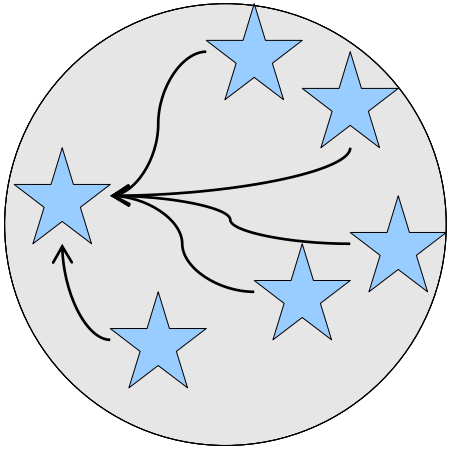


P = NP

The Tantalizing Truth

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

Proof: Suppose that L is **NP**-complete and $L \in \mathbf{P}$. 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 $\mathbf{P} = \mathbf{NP}$. ■



P = NP

The Tantalizing Truth

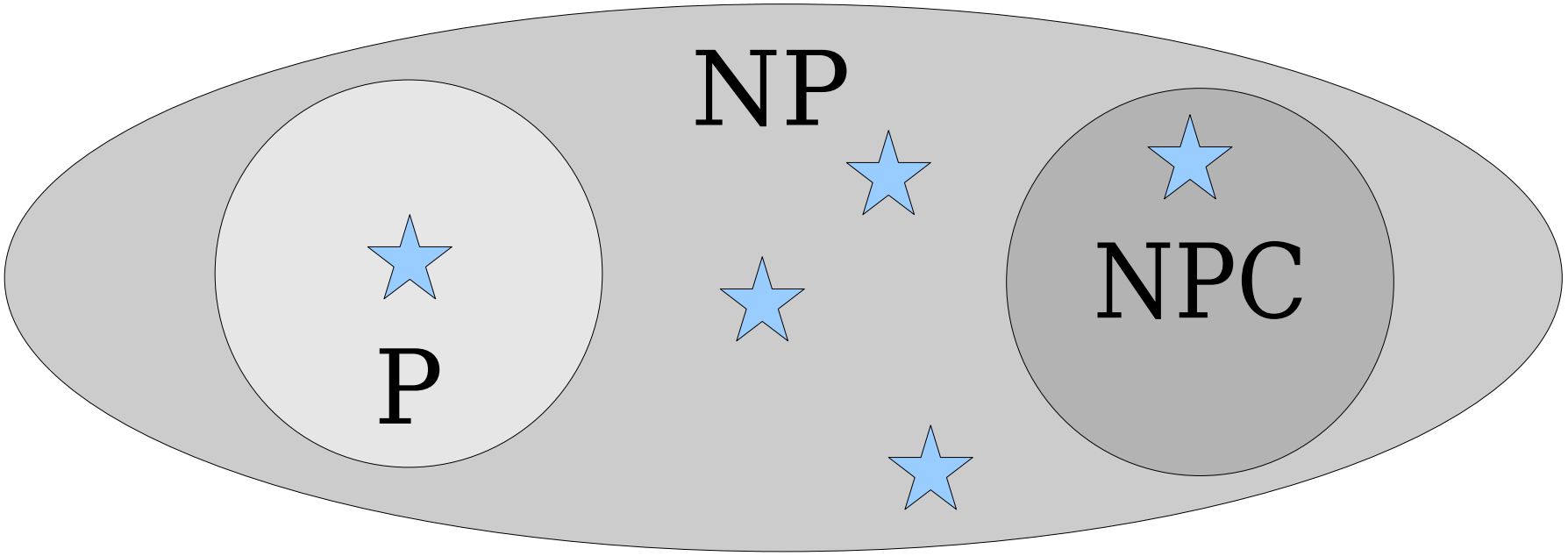
Theorem: If *any* **NP**-complete language is not in **P**, then **P ≠ NP**.

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

The Tantalizing Truth

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

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



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

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

Proof Idea: To see that **SAT** ∈ **NP**, show how to make a polynomial-time verifier for it. Key idea: have the certificate be a satisfying assignment.

To show that **SAT** is **NP**-hard, given a polymomial-time verifier V for an arbitrary **NP** language L, for any string w you can construct a polynomially-sized formula $\varphi(w)$ that says "there is a certificate c where V accepts $\langle w, c \rangle$." This formula is satisfiable if and only if $w \in L$, so deciding whether the formula is satisfiable decides whether w is in L. ■-*ish*

Proof: Take CS154!

Why All This Matters

Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is equivalent to just figuring out how hard SAT is.

$$\text{SAT} \in \mathbf{P} \quad \leftrightarrow \quad \mathbf{P = NP}$$

We've turned a huge, abstract, theoretical problem about solving problems versus checking solutions into the concrete task of seeing how hard one problem is.

You can get a sense for how little we know about algorithms and computation given that we can't yet answer this question!

Why All This Matters

You will almost certainly encounter **NP**-hard problems in practice – they're everywhere!

If a problem is **NP**-hard, then there is no known algorithm for that problem that

- is efficient on all inputs,
- always gives back the right answer, and
- runs deterministically.

Useful intuition: If you need to solve an **NP**-hard problem, you will either need to settle for an approximate answer, an answer that's likely but not necessarily right, or have to work on really small inputs.

Sample **NP**-Hard Problems

Computational biology: Given a set of genomes, what is the most probable evolutionary tree that would give rise to those genomes? *(Maximum parsimony problem)*

Game theory: Given an arbitrary perfect-information, finite, two-player game, who wins? *(Generalized geography problem)*

Operations research: Given a set of jobs and workers who can perform those tasks in parallel, can you complete all the jobs within some time bound? *(Job scheduling problem)*

Machine learning: Given a set of data, find the simplest way of modeling the statistical patterns in that data *(Bayesian network inference problem)*

Medicine: Given a group of people who need kidneys and a group of kidney donors, find the maximum number of people who survive. *(Cycle cover problem)*

Systems: Given a set of processes and a number of processors, find the optimal way to assign those tasks so that they complete as soon as possible *(Processor scheduling problem)*

Coda: What if $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is resolved?

Time-Out for Announcements!

Please evaluate this course on Axess.

Your feedback really makes a difference.

The Big Picture

Take a minute to reflect on your journey.

Set Theory

Power Sets

Cantor’s Theorem

Direct Proofs

Parity

Proof by Contrapositive

Proof by Contradiction

Modular Congruence

Number Theory

Propositional Logic

First-Order Logic

Logic Translations

Logical Negations

Propositional Completeness

Vacuous Truths

Tournament Graphs

Binary Relations

Equivalence Relations

Equivalence Classes

Systems of Representatives

Strict Orders

Functions

Injections

Surjections

Bijections

Inverse Functions

Permutations

Graphs

Connectivity

Graph Automorphisms

Vertex Covers

Bipartite Graphs

Mathematical Induction

Loop Invariants

Complete Induction

Tiling Problems

Bezout's Identity

Euclid's Algorithm

Hypercubes

Formal Languages

DFAs

Regular Languages

Closure Properties

NFAs

Subset Construction

Kleene Closures

Monoids

5-Tuples

Regular Expressions

State Elimination

Distinguishability

Myhill-Nerode Theorem

Nonregular Languages

Extended Transition Functions

Equivalence Relation Indices

Axiom of Choice

Context-Free Grammars

Turing Machines

Church-Turing Thesis

TM Encodings

Universal Turing Machines

Self-Reference

Decidability

Recognizability

Self-Defeating Objects

Undecidable Problems

Halting Problem

Verifiers

Diagonalization Language

Complexity Class **P**

Complexity Class **NP**

P $\stackrel{?}{=}$ **NP** Problem

Polynomial-Time Reducibility

NP-Completeness

You’ve done more than just check a bunch of boxes off a list.

You've given yourself the foundation to tackle problems from all over computer science.

From CS255

A **Shannon cipher** is a pair $\mathcal{E} = (E, D)$ of functions.

- The function E (the **encryption function**) takes as input a **key** k and a **message** m (also called a **plaintext**), and produces as output a **ciphertext** c. That is,

  $$c = E(k, m),$$

  and we say that c is the **encryption of** m **under** k.

- The function D (the **decryption function**) takes as input a key k and a ciphertext c, and produces a message m. That is,

  $$m = D(k, c),$$

  and we say that m is the **decryption of** c **under** k.

- We require that decryption "undoes" encryption; that is, the cipher must satisfy the following **correctness property**: for all keys k and all messages m, we have

  $$D(k,\ E(k,\ m)\ ) = m.$$

Kinda sorta like a left inverse!

To be slightly more formal, let us assume that $\mathcal{K}$ is the set of all keys (the **key space**), $\mathcal{M}$ is the set of all messages (the **message space**), and that $\mathcal{C}$ is the set of all ciphertexts (the **ciphertext space**). With this notation, we can write:

$$E : \mathcal{K} \times \mathcal{M} \to \mathcal{C},$$
$$D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}.$$

Hey, you've seen this before!

Also, we shall say that $\mathcal{E}$ is **defined over** $(\mathcal{K}, \mathcal{M}, \mathcal{C})$.

From CS145

Semantics of JOINs (2 tables)

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

Cartesian products!

1. Take **cross product**:
   $X = R \times S$

Recall: Cross product (A X B) is the set of all unique tuples in A,B
Ex: {a,b,c} X {1,2}
= {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

2. Apply **selections / conditions**:
   $Y = \{(r,s) \in X \mid r.A == r.B\}$

= Filtering!

Set-builder notation!

3. Apply **projections** to get final output:
   $Z = (y.A,) \text{ for } y \in Y$

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on…)

From CS251

CS251: Cryptocurrencies and Blockchain Technologies **Fall 2018**

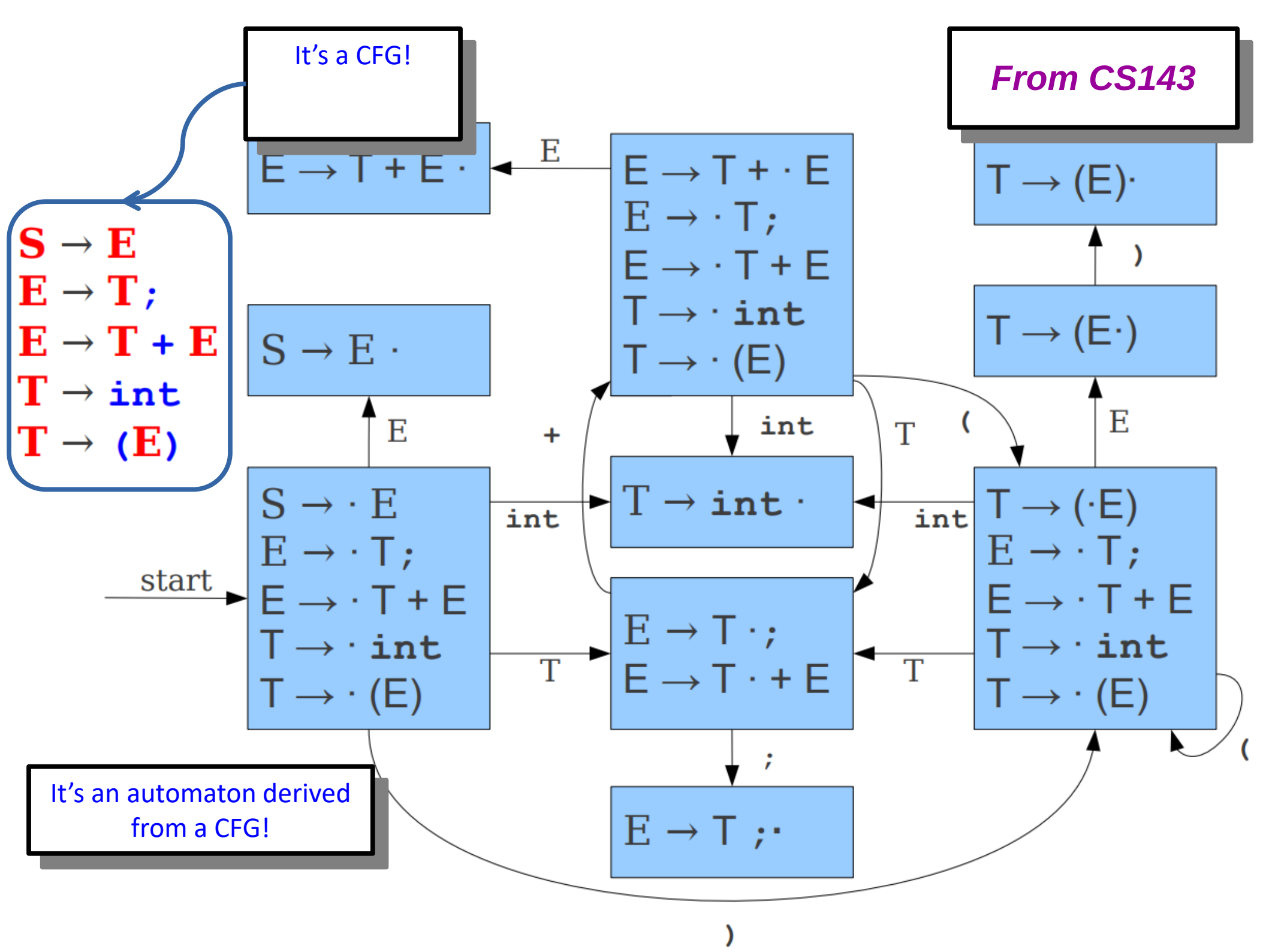
Assignment #1

Due: 11:59pm on Mon., **Oct. 8, 2018**
Submit via Gradescope (each answer on a separate page) code: **9RZGVZ**

Problem 1. Hash functions and proofs of work. In class we defined two security properties for a hash function, one called collision resistance and the other called proof-of-work security. Show that a collision-resistant hash function may not be proof-of-work secure.

Hint: let $H : X \times Y \to \{0, 1, \ldots, 2^n - 1\}$ be a collision-resistant hash function. Construct a new hash function $H' : X \times Y \to \{0, 1, \ldots, 2^m - 1\}$ (where m may be greater than n) that is also collision resistant, but for a fixed difficulty D (say, $D = 2^{32}$) is not proof-of-work secure with difficulty D. That is, for every puzzle $x \in X$ it should be trivial to find a solution $y \in Y$ such that $H'(x, y) < 2^m/D$. This is despite H' being collision resistant. Remember to explain why your H' is collision resistant, that is, explain why a collision on H' would yield a collision on H.

Whoa, it's a function!


It's a CFG!
From CS143
S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
S → E ·
T → (E·)
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
E → T ·;
E → T · + E
T
T
;
E → T ;·
(
)
It's an automaton derived from a CFG!

Search problems

From CS221

Definition: search problem

States: the set of states

$s_{\text{start}} \in \text{States}$: starting state

$\text{Actions}(s)$: possible actions from state s

$\text{Succ}(s, a)$: where we end up if take action a in state s

$\text{Cost}(s, a)$: cost for taking action a in state s

$\text{IsEnd}(s)$: whether at end

- $\text{Succ}(s, a) \Rightarrow T(s, a, s')$
- $\text{Cost}(s, a) \Rightarrow \text{Reward}(s, a, s')$

It's a DFA!

pronounced “big-oh of …” or sometimes “oh of …”

From CS161

O(…) means an upper bound

- Let T(n), g(n) be functions of positive integers.
  - Think of T(n) as being a runtime: positive and increasing in n.

- We say “T(n) is O(g(n))” if g(n) grows at least as fast as T(n) as n gets large.

- Formally,

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$

It’s FOL and functions!

From CS224W

- Graph $G(V, E)$ has **expansion α**: if $\forall S \subseteq V$:

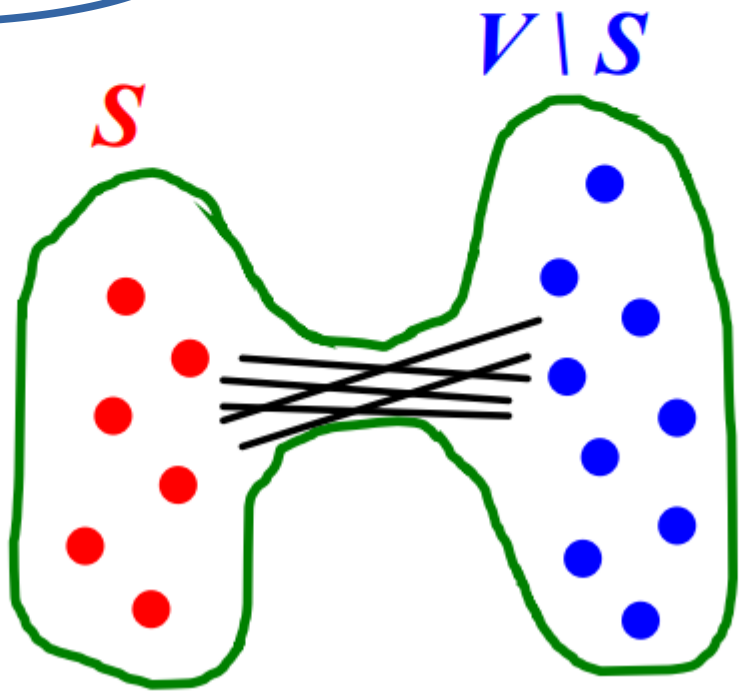
  $\#\text{ of edges leaving } S \geq \alpha \cdot \min(|S|, |V \setminus S|)$

- **Or equivalently:**

$$\alpha = \min_{S \subseteq V} \frac{\#\,edges\ leaving\ S}{\min(|S|, |V \setminus S|)}$$

Set difference and cardinality!

First-order definitions on graphs!



From CS242

Typed lambda calculus

To understand the formal concept of a type system, we're going to extend our lambda calculus from last week (henceforth the "untyped" lambda calculus) with a notion of types (the "simply typed" lambda calculus). Here's the essentials of the language:

Type τ ::=	int	integer
|	$\tau_1 \to \tau_2$	function
Expression e ::=	x	variable
|	n	integer
|	$e_1 \oplus e_2$	binary operation
|	$\lambda\ (x : \tau)\ .\ e$	function
|	$e_1\ e_2$	application
Binop $\oplus$ ::=	$+ \mid - \mid * \mid /$	

It's a CFG!

First, we introduce a language of types, indicated by the variable tau (τ). A type is either an integer, or a function from an input type τ_1 to an output type τ_2. Then we extend our untyped lambda calculus with the same arithmetic language from the first lecture (numbers and binary operators)[4]. Usage of the language looks similar to before:

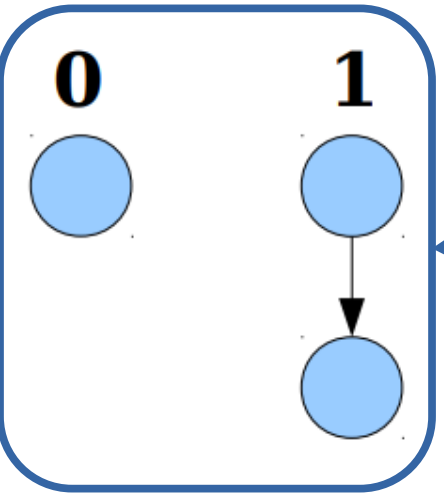
From CS166

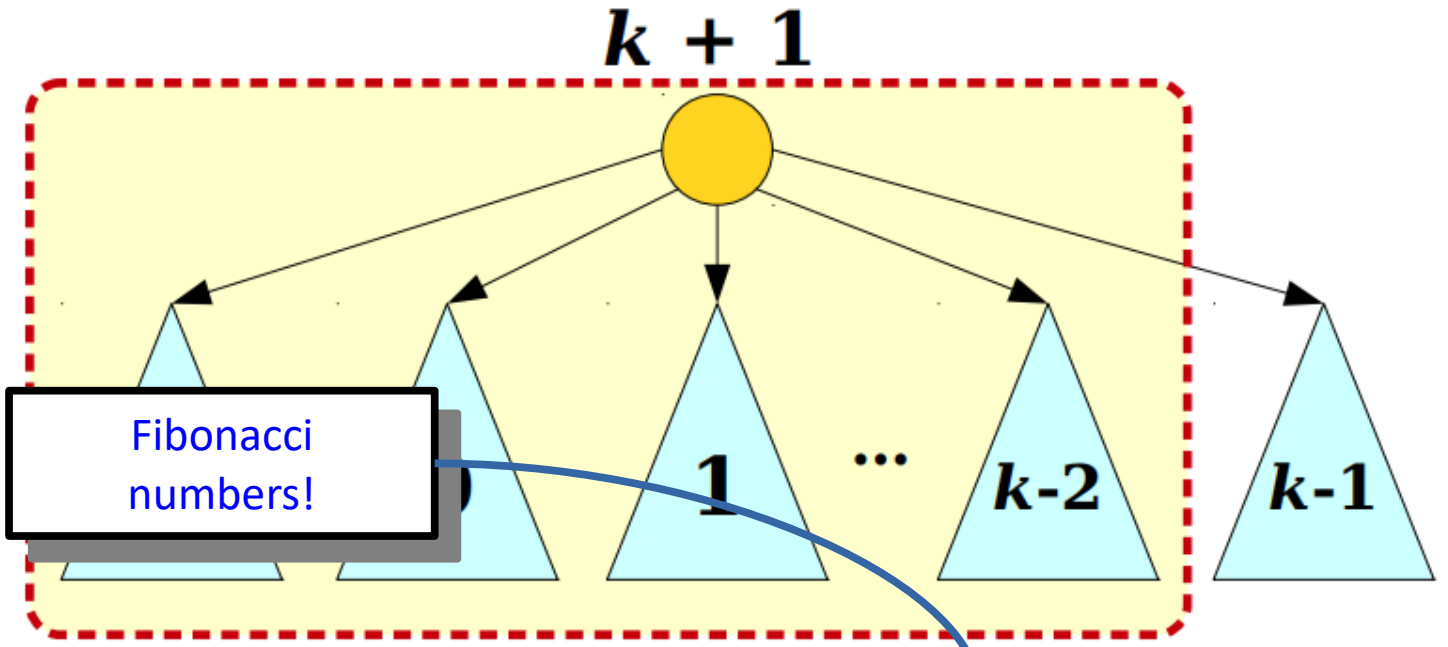
- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.

- ***Proof:*** Induction.

Formal proofs!

Trees!





Fibonacci numbers!

F_2 F_3 F_{k+2} + F_{k+1}

From CS238

3.1.1 Constraints on Rational Preferences

Just as we imposed a set of constraints on beliefs, we will impose some constraints on preferences. These constraints are sometimes called the *von Neumann-Morgenstern axioms*, named after John von Neumann and Oskar Morgenstern, who formulated a variation of these axioms in the 1940s.

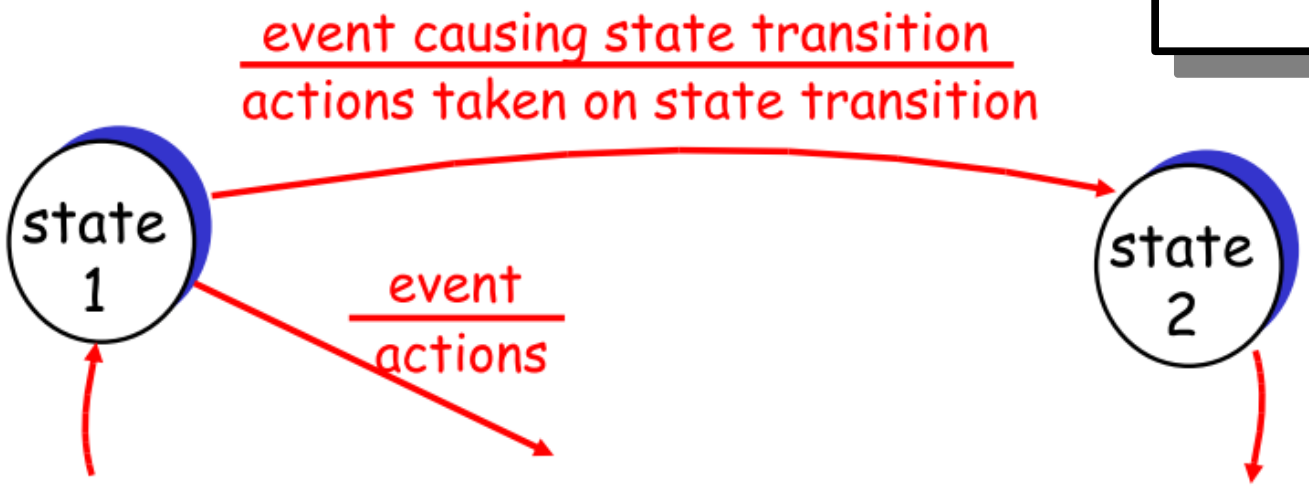
- *Completeness.* Exactly one of the following hold: $A \succ B$, $B \succ A$, or $A \sim B$.
- *Transitivity.* If $A \succeq B$ and $B \succeq C$, then $A \succeq C$.
- *Continuity.* If $A \succeq C \succeq B$, then there exists a probability p such that $[A : p; B : 1 - p] \sim C$.
- *Independence.* If $A \succ B$, then for any C and probability p, $[A : p; C : 1 - p] \succeq [B : p; C : 1 - p]$.

These are constraints on *rational preferences*. They say nothing about the preferences of actu fact, there is strong evidence that humans are not very rational (Sec jective in this book is to understand rational decision making from a computational perspective so that we can build useful systems. The possible extension of this theory to understanding human decision making is of only secondary interest.

Hey, we know that one!

Finite State Machines

From CS144



- **Represent protocols using state machines**
  - Sender and receiver each have a state
  - Start in some initial state
  - Events cause each side to select a state transition

It's a generalization of DFAs!

- **Transition specifies action taken**
  - Specified as events/actions
  - E.g., software calls `send`/put packet on network
  - E.g., packet arrives/send acknowledgment

From CS168

Reducibility!

By definition, we need to output y if and only if $y \in S$. That is, *answering membership queries reduces to solving the Heavy Hitters problem.* By the "membership problem," we mean the task of preprocessing a set S to answer queries of the form "is $y \in S$"? (A hash table is the most common solution to this problem.) It is intuitive that you cannot correctly answer all membership queries for a set S without storing S (thereby using linear, rather than constant, space) — if you throw some of S out, you might get a query asking about the part you threw out, and you won't know the answer. It's not too hard to make this idea precise using the Pigeonhole Principle.[5]

A Myhill-Nerode-style argument!

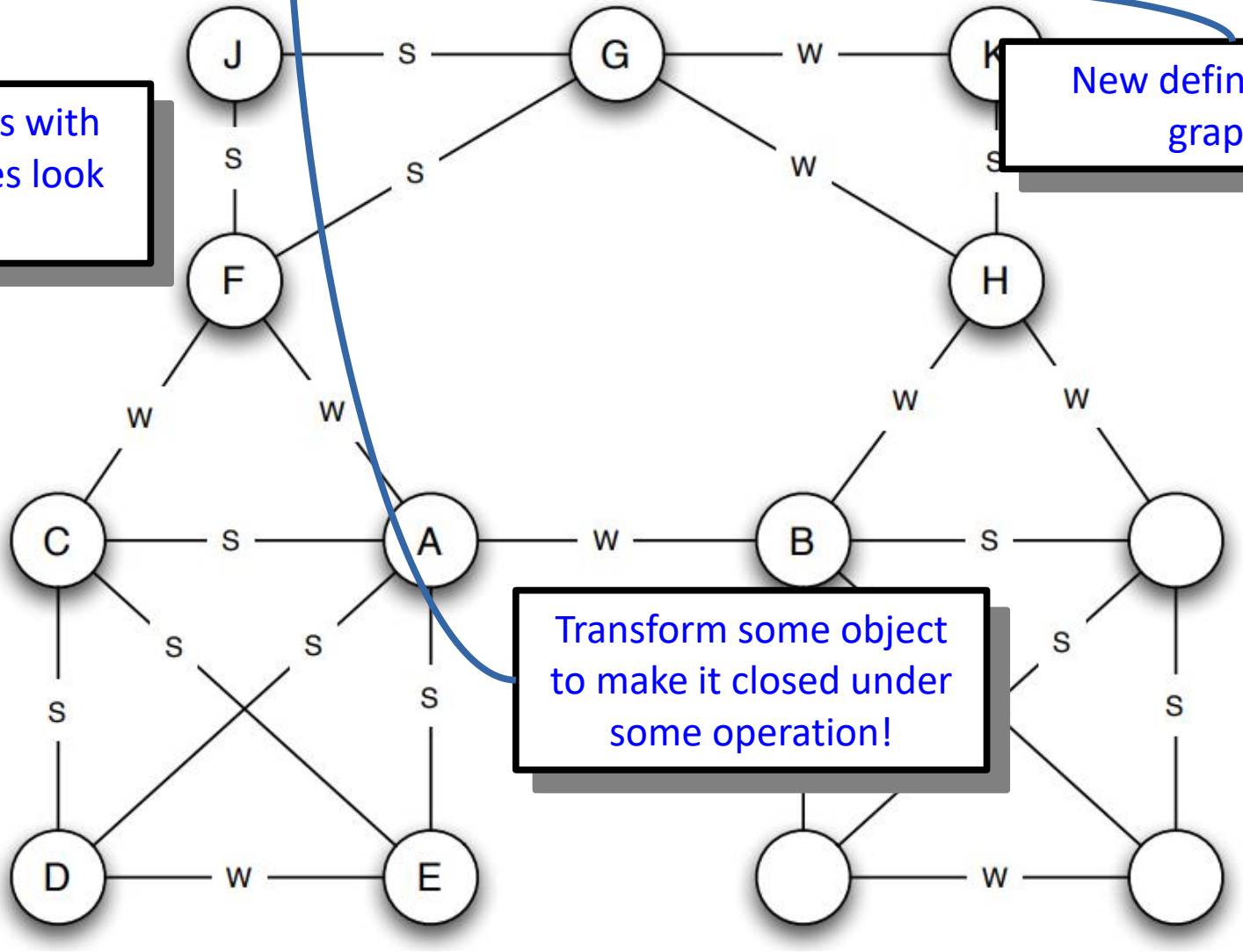
Strong triadic closure

From CS124

If a node Q has two strong ties to nodes Y and Z, there is an edge between Y and Z

What do graphs with these properties look like?

New definitions on graphs!



Transform some object to make it closed under some operation!

From CS154

Kolmogorov Complexity (1960's)

Definition: The *shortest description of* x, denoted as $d(x)$, is the lexicographically shortest string $\langle M,w \rangle$ such that $M(w)$ halts with only x on its tape.

Definition: The *Kolmogorov complexity of* x, denoted as $K(x)$, is $|d(x)|$.

Using Turing machines to define intrinsic information content!

From CS246

- **Suppose we are given a set of documents D**
  - Each document **d** covers a set X_d of words/topics/named entities **W**
- **For a set of documents $A \subseteq D$ we define**

$$F(A) = \left| \bigcup_{d \in A} X_d \right|$$

Functions, set union, and set cardinality!

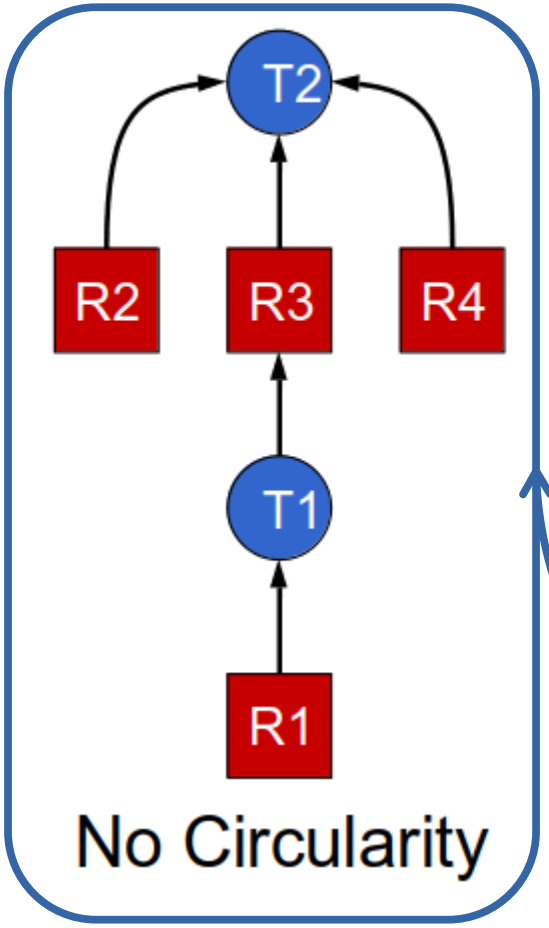
- **Goal: We want to**

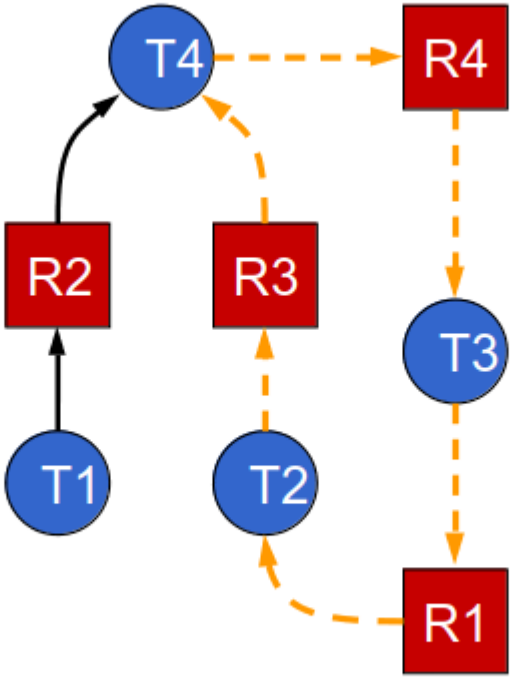
$$\max_{|A| \leq k} F(A)$$

- **Note: *F(A)* is a set function: $F(A)$: Sets $\rightarrow \mathbb{N}$**

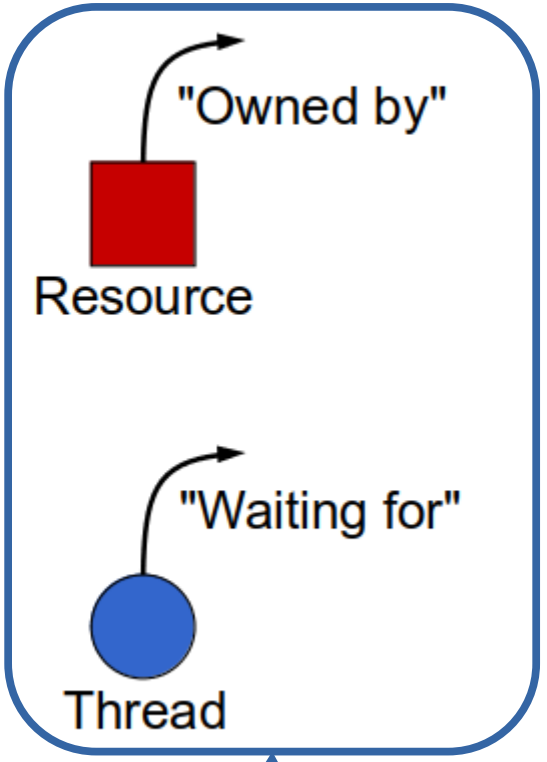
From CS140

Circular Requests





This is a strict order!



These are binary relations!

You've given yourself the foundation to tackle problems from all over computer science.

There’s so much more to explore.

Where should you go next?

Course Recommendations

Theoryland

- CS154
- Phil 151
- Phil 152 — *Computability*
- Math 107
- Math 108 — *Graphs*
- Math 120 — *Symmetries*
- Math 113 — *Functions*
- Math 161 — *Set Theory*
- Math 152 — *Number Theory*

Applications

- CS124 — *Languages / Automata*
- CS143 — *Languages / Automata*
- CS161 — *Graphs*
- CS224W — *Graphs*
- CS243 — *Relations*
- CS246 — *Functions*
- CS242 — *Functions*
- CS251 — *Functions*
- CS255 — *Functions*

Your Questions

“As computers become more ubiquitous, it seems that knowledge about computers and how they work does not. How can we address this? Does LinkedIn have a perspective on this? Your personal thoughts?”

Though there’s still a lot we don’t know about computation, the amount that we do know is increasing every day! CS research (theoretical and applied) is constantly discovering new things about what’s possible with a computer and new algorithms, new applications of existing techniques, etc.

“If I get an awful grade on the final exam, is it possible to still do okay-ish in the class (if I've been keeping up with the problem sets)?”

I don’t think this is the right question to be asking right now: 1) If you’ve been keeping up with the problem sets, there’s no reason to expect that you’ll do poorly on the exam, and 2) the final hasn’t happened yet so the outcome is still very much in your hands! Instead of thinking of what could happen if the exam goes badly, focus your energy on actively targeting your weak spots and working to make the exam go well.

"What is the high-level knowledge map for math used in CS? PSets do give a glimpse into how much math is involved in many areas of CS but what is the big picture?"

We saw some specific examples earlier today, here's a broader overview*

* Huge caveat: this is largely based on which areas of CS I've personally been exposed to! There's lots that I don't have much experience in.

AI

Networking

NLP

Compilers

Databases

Crypto

Information Theory

Set theory

Logic

Probability

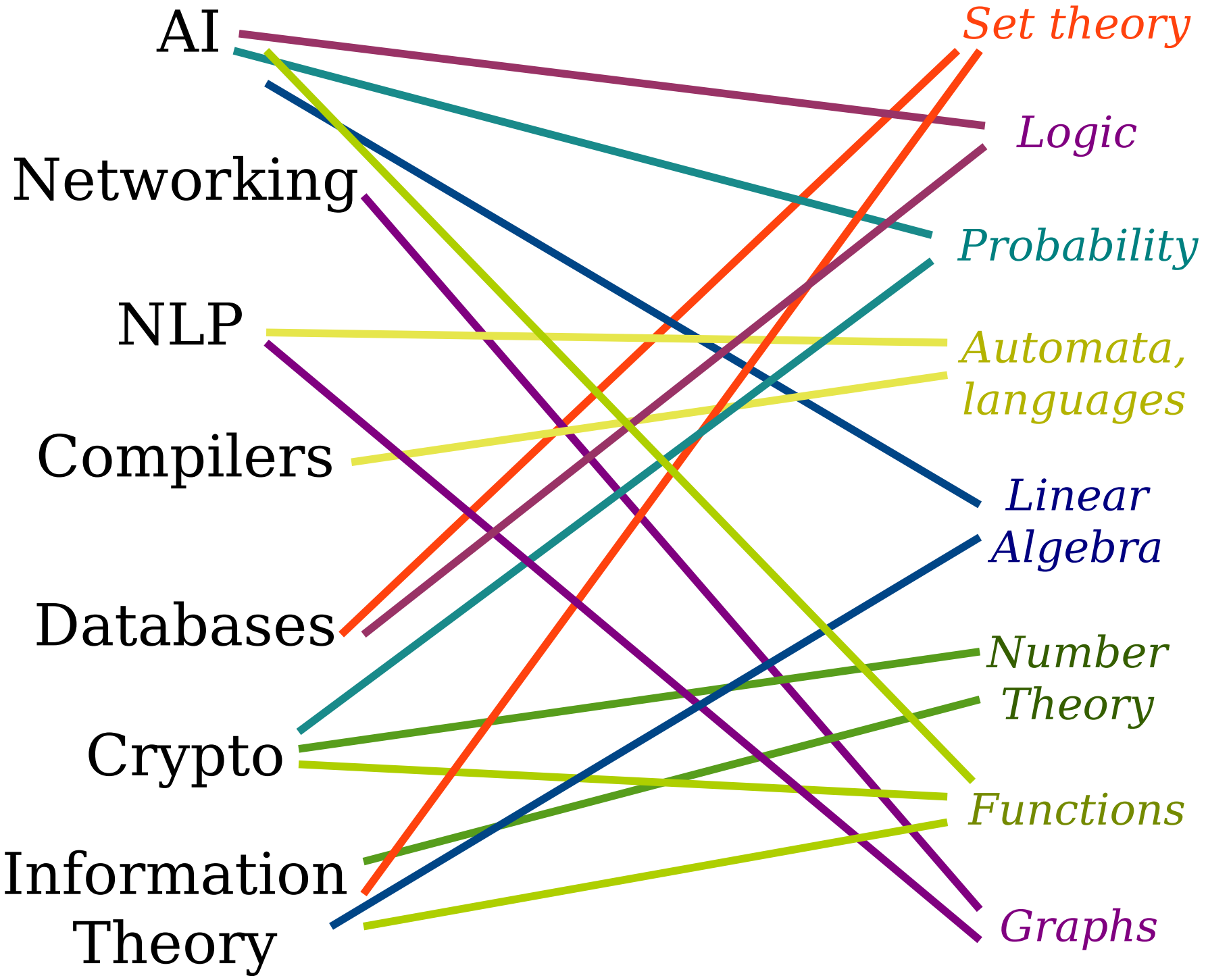
Automata, languages

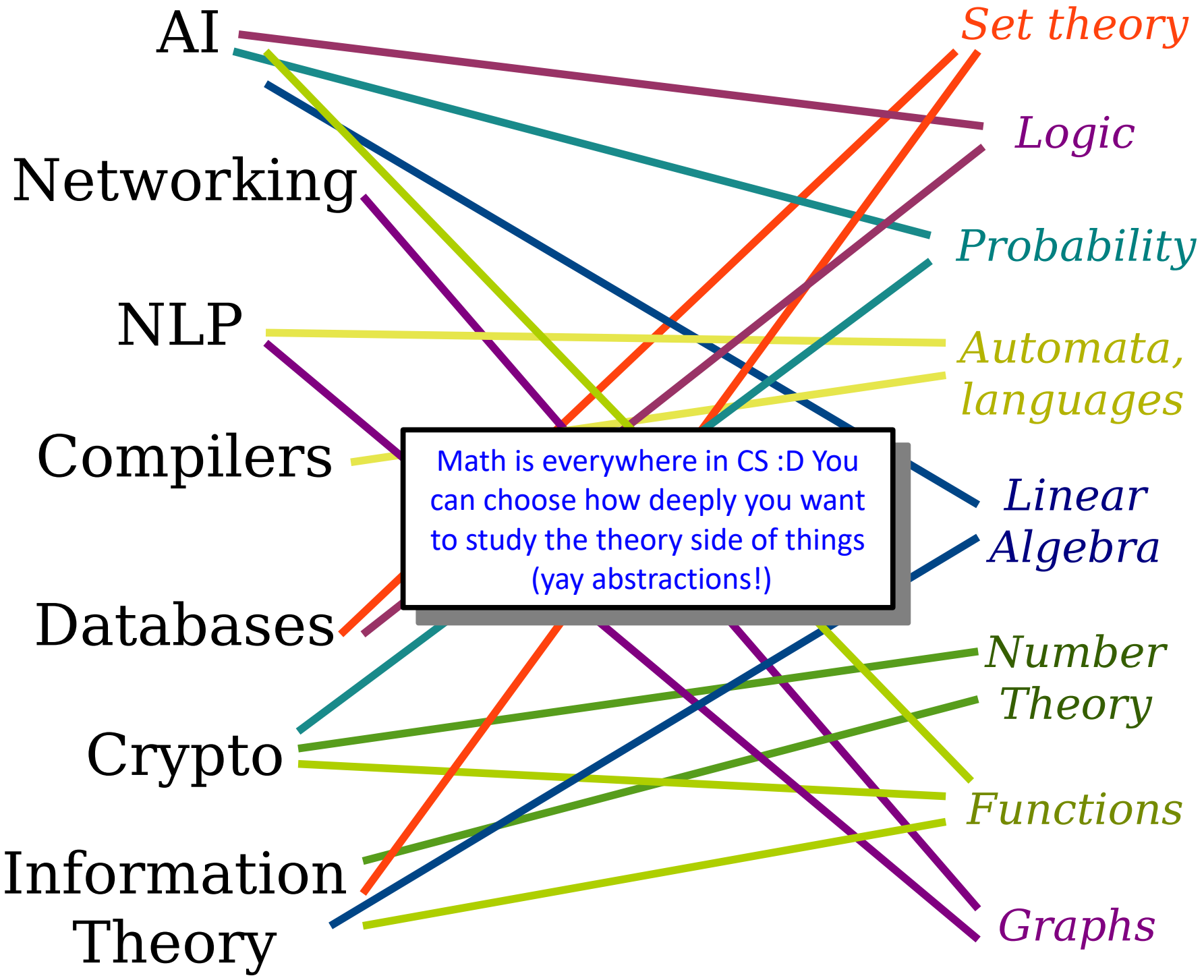
Linear Algebra

Number Theory

Functions

Graphs


AI
Networking
NLP
Compilers
Databases
Crypto
Information Theory
Set theory
Logic
Probability
Automata, languages
Linear Algebra
Number Theory
Functions
Graphs


AI
Networking
NLP
Compilers
Databases
Crypto
Information Theory
Set theory
Logic
Probability
Automata, languages
Linear Algebra
Number Theory
Functions
Graphs
Math is everywhere in CS :D You can choose how deeply you want to study the theory side of things (yay abstractions!)

“What does doing research in CS look like? Like for bench science, we will conduct experiments to test an idea. It's hard to imagine CS researches as sitting in an office and coding all day long.”

Similar process, just different tools! CS research can take many forms: coming up with new ways of modeling and predicting phenomena, designing new algorithms and proving that they meet certain runtime/space constraints, finding new ways of applying CS to other fields (education, healthcare, transportation, …) Would be happy to connect you to folks here in the department who are doing CS research!

“Out of all the CS courses offered at Stanford…why CS103? What makes this course more compelling to you than others?”

:)

We get to take these abstract, complex, philosophical ideas (the nature of computation, infinity, truth) and make them accessible and tangible.

As a class in the CS core @ Stanford, CS103 presents the unique opportunity to get people excited about computer science from a completely different perspective.

Final Thoughts

A Huge Round of Thanks!

There are more problems to solve than there are programs capable of solving them.

There is so much more to explore and so many big questions to ask – ***many of which haven't been asked yet!***


Theory
Practice

You now know what problems we can solve, what problems we can't solve, and what problems we believe we can't solve efficiently.

Our questions to you:

What problems will you ***choose*** to solve?

Why do those problems matter to you?

And how are you going to solve them?