Complexity Theory

Part One

Up to this point:

“*Can* we solve this problem?”

(**Computability Theory**)

Starting today:

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

(**Complexity Theory**)

A Decidable Problem

- ***Presburger arithmetic*** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \to x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $(P(0) \land \forall y.\ (P(y) \to P(y + 1))) \to \forall x.\ P(x)$
- 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.

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

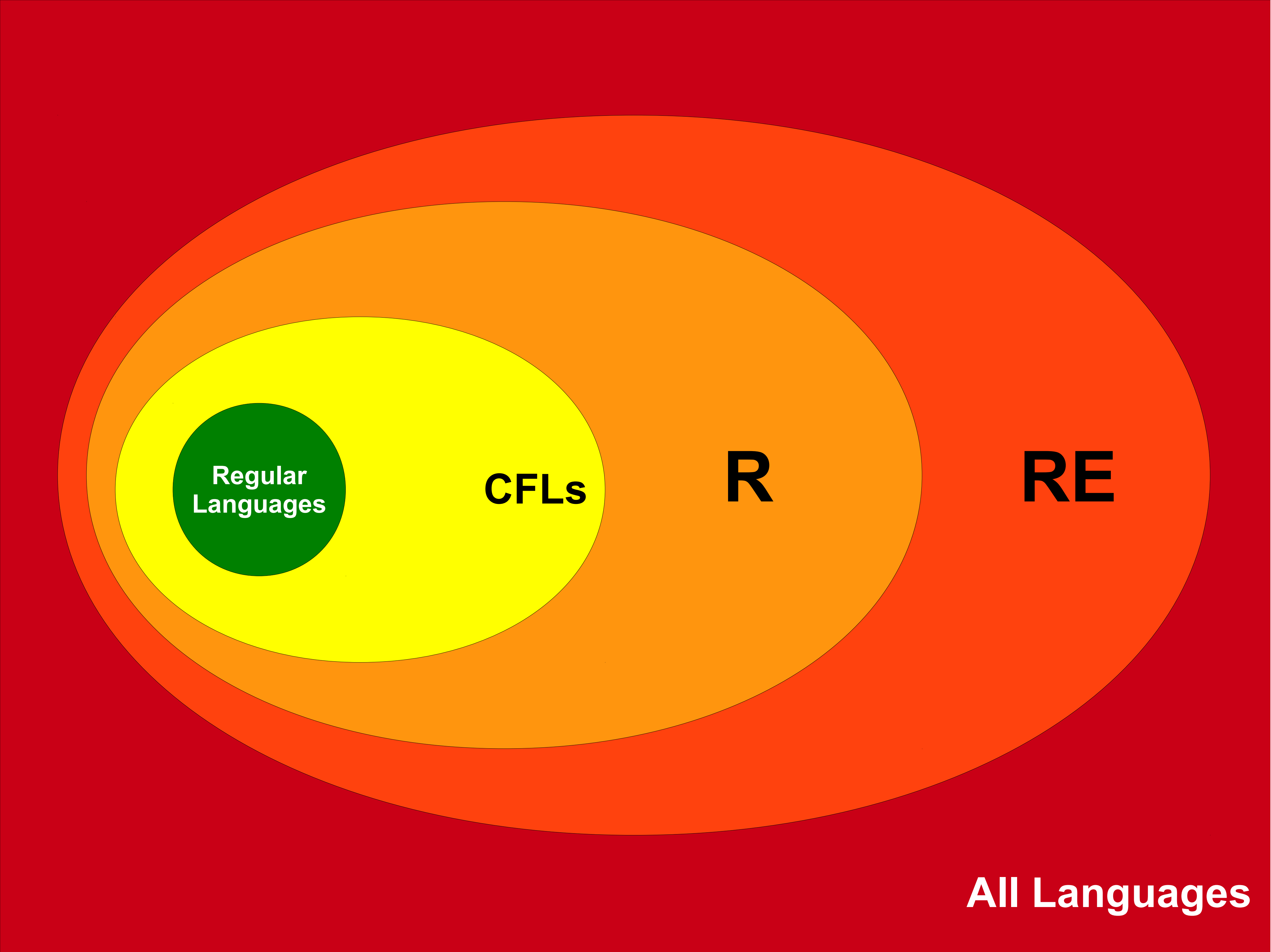
- In the … l explor…

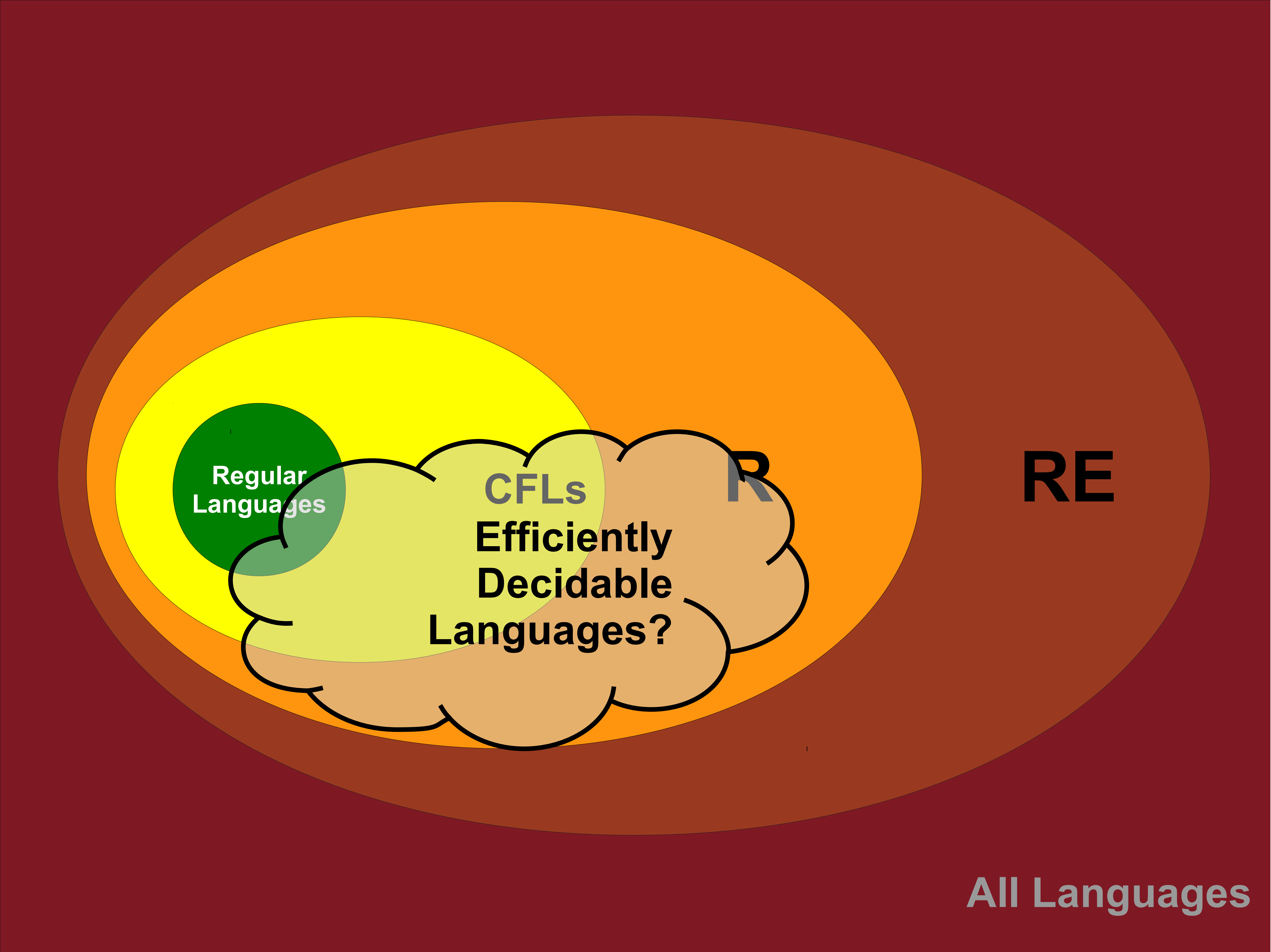
Where We've Been

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

Where We're Going

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


Regular Languages
CFLs
R
RE
All Languages


Regular Languages
CFLs
Efficiently Decidable Languages?
R
RE
All Languages

The Setup

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

Measuring Complexity

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

Measuring Complexity

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

Measuring Complexity

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

What is an efficient algorithm?

Searching Finite Spaces

- Many decidable problems can be solved by searching over a large but finite space of possible options.
- Searching this space might take a staggeringly long time, but only finite time.
- From a decidability perspective, this is totally fine.
- From a complexity perspective, this may be totally unacceptable.

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

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

2.4s

Easy!

Traveling Salesperson Problem:
We have a bunch of cities to visit. In what order should we visit them to minimize total travel distance?

Traveling Salesperson Problem:

We have a bunch of cities to visit. In what order should we visit them to minimize total travel distance?

Exhaustively try all orderings: **O(n!)**

Use current best known algorithm: **O(n²2ⁿ)**

Maybe we could invent an algorithm that fits in our rightmost column: **O(2ⁿ)**

So <u>let's say</u> we come up with a way to solve Traveling Salesperson Problem in O(2^n).

It would take 4 days to solve Traveling Salesperson Problem on 50 state capitals.

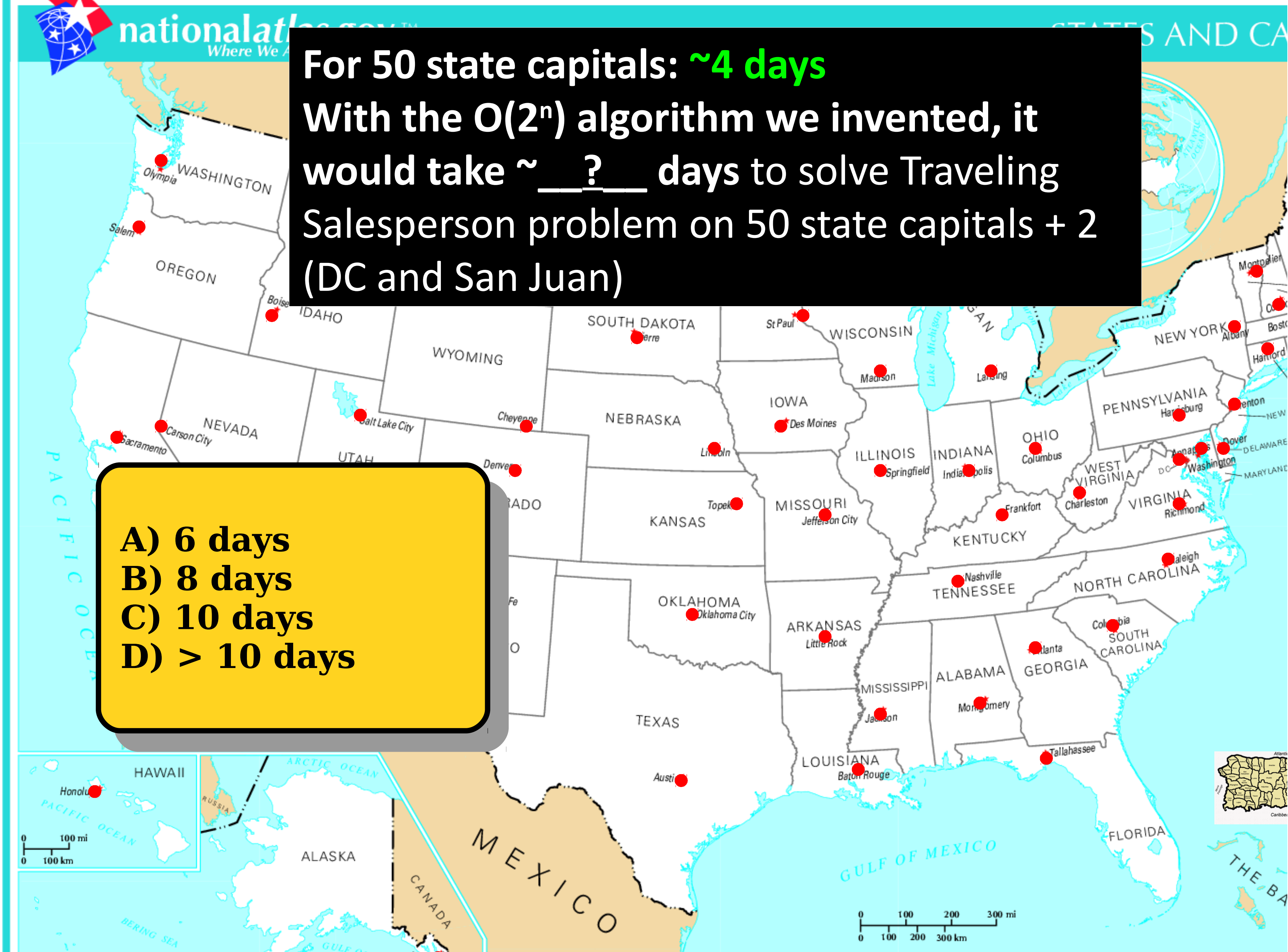
Two *tiny* little updates

- Imagine we approve statehood for Puerto Rico
  - Add San Juan, the capital city
- Also add Washington, DC

- **Now 52 capital cities instead of 50**

For 50 state capitals: ~4 days

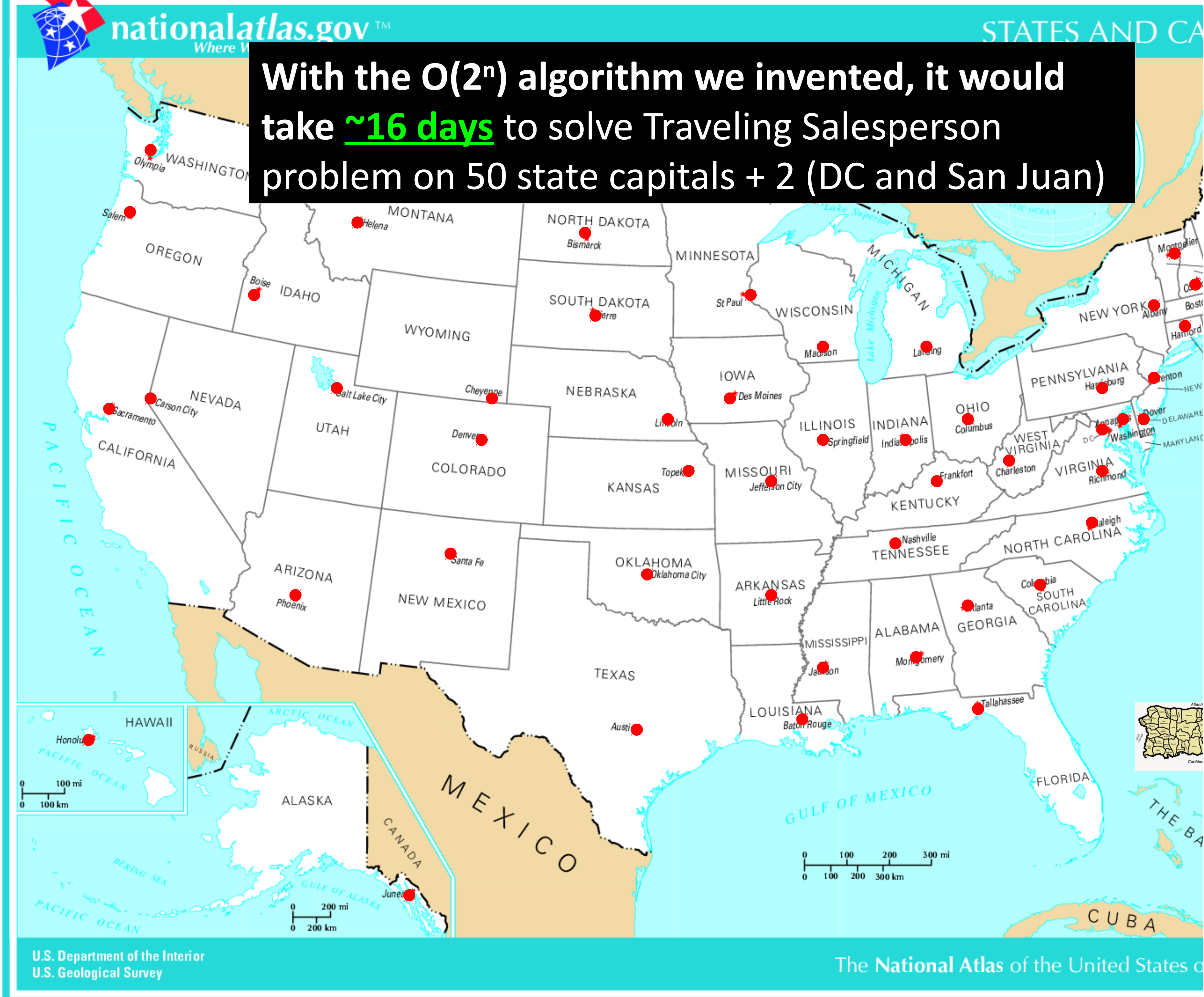
With the $O(2^n)$ algorithm we invented, it would take ~__?__ days to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)



A) 6 days
B) 8 days
C) 10 days
D) > 10 days

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **Y** or **N**.

With the $O(2^n)$ algorithm we invented, it would take <u>~16 days</u> to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)

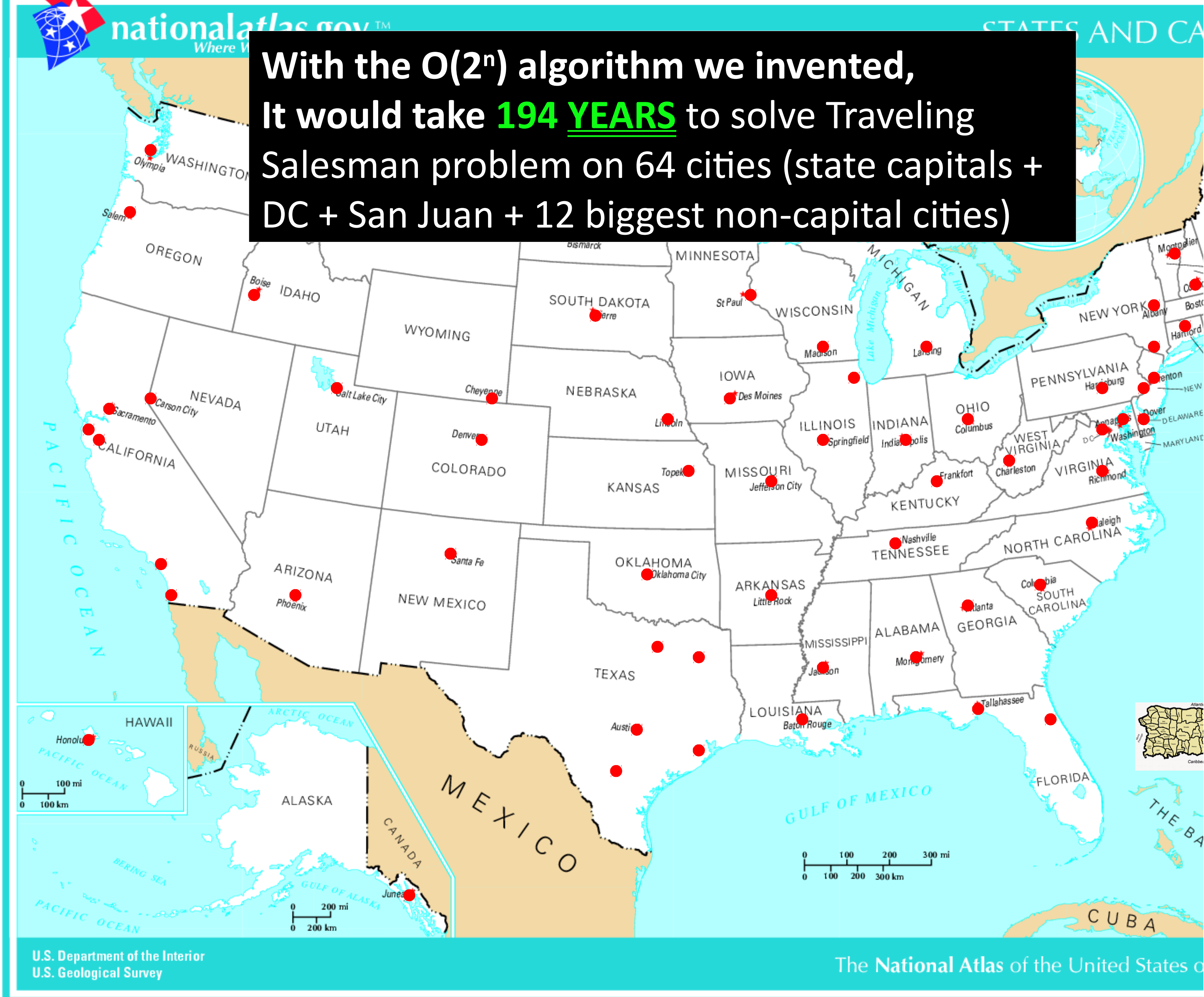


Sacramento is not exactly the most interesting or important city in California (sorry, Sacramento).

What if we add the 12 biggest non-capital cities in the United States to our map?

With the $O(2^n)$ algorithm we invented,

It would take 194 YEARS to solve Traveling Salesman problem on 64 cities (state capitals + DC + San Juan + 12 biggest non-capital cities)



$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**			194 **YEARS**
8	**256**			
9	**512**			
10	**1,024**			
30	**2,070,000,000**			

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

3.59E+21 **YEARS**

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

3,590,000,000,000,000,000,000
YEARS

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

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

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

1.42E+137 **YEARS**

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

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

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

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

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

A Sample Problem

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

A Sample Problem

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

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

A Sample Problem

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

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

A Sample Problem

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

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

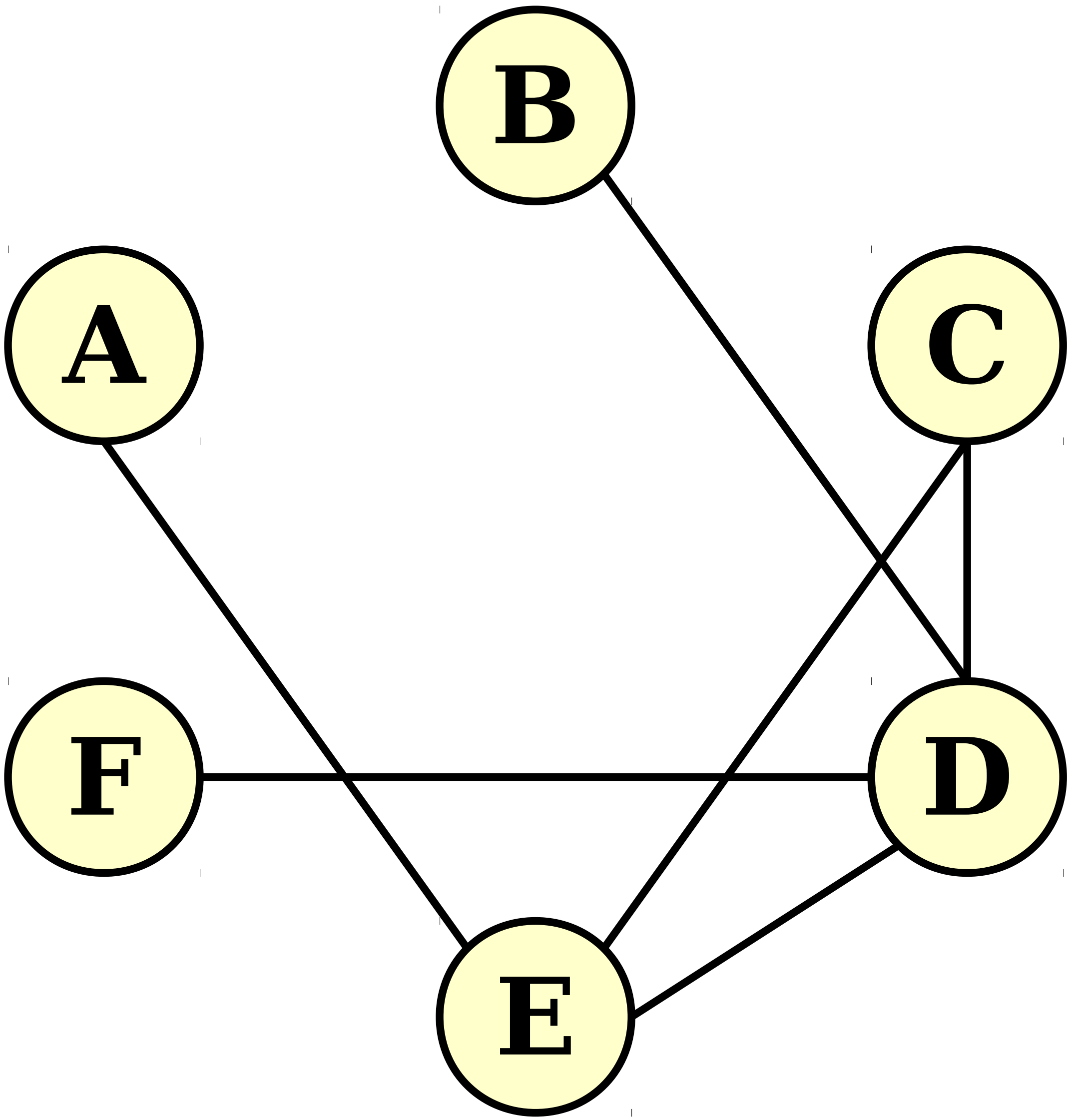
Longest Increasing Subsequences

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

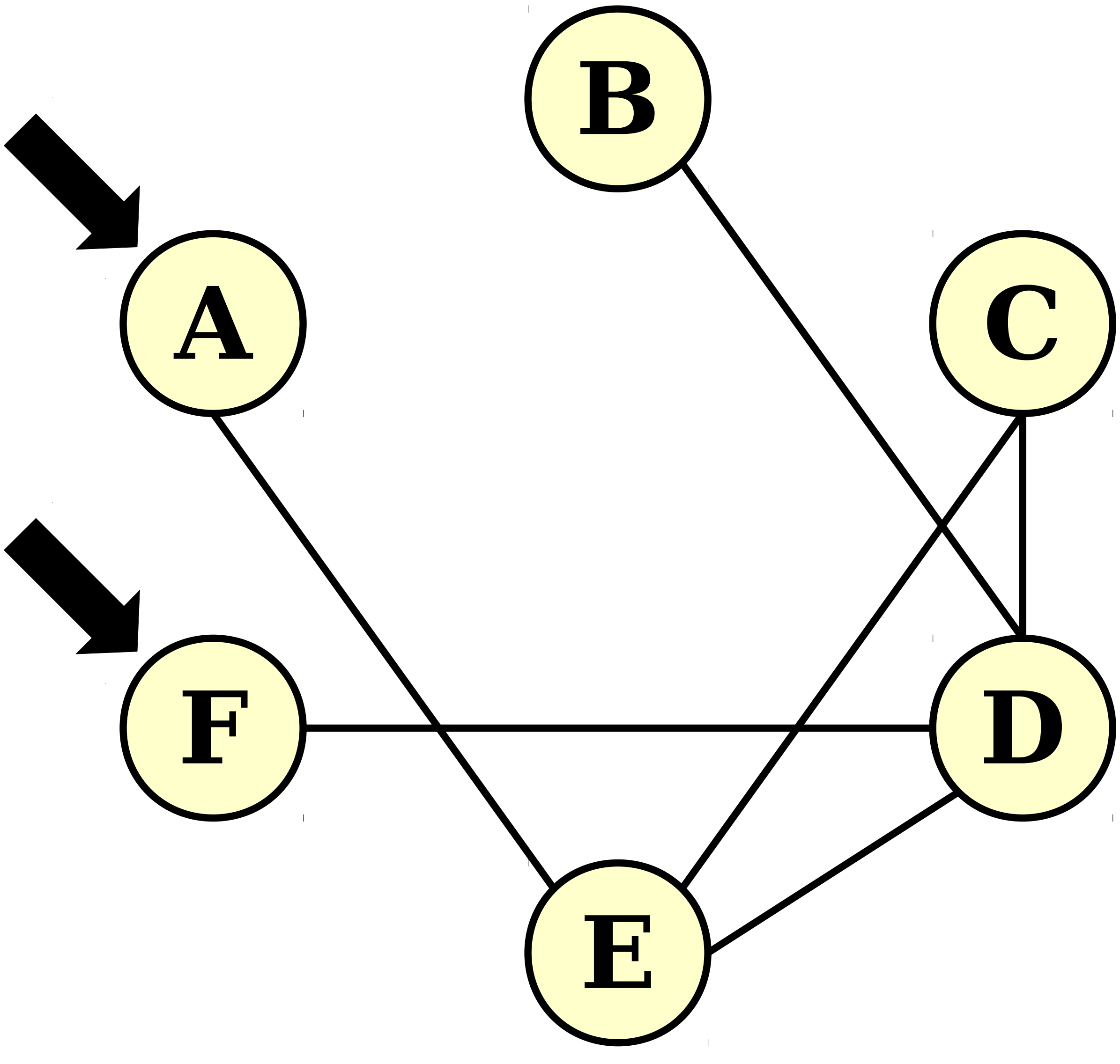
Longest Increasing Subsequences

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

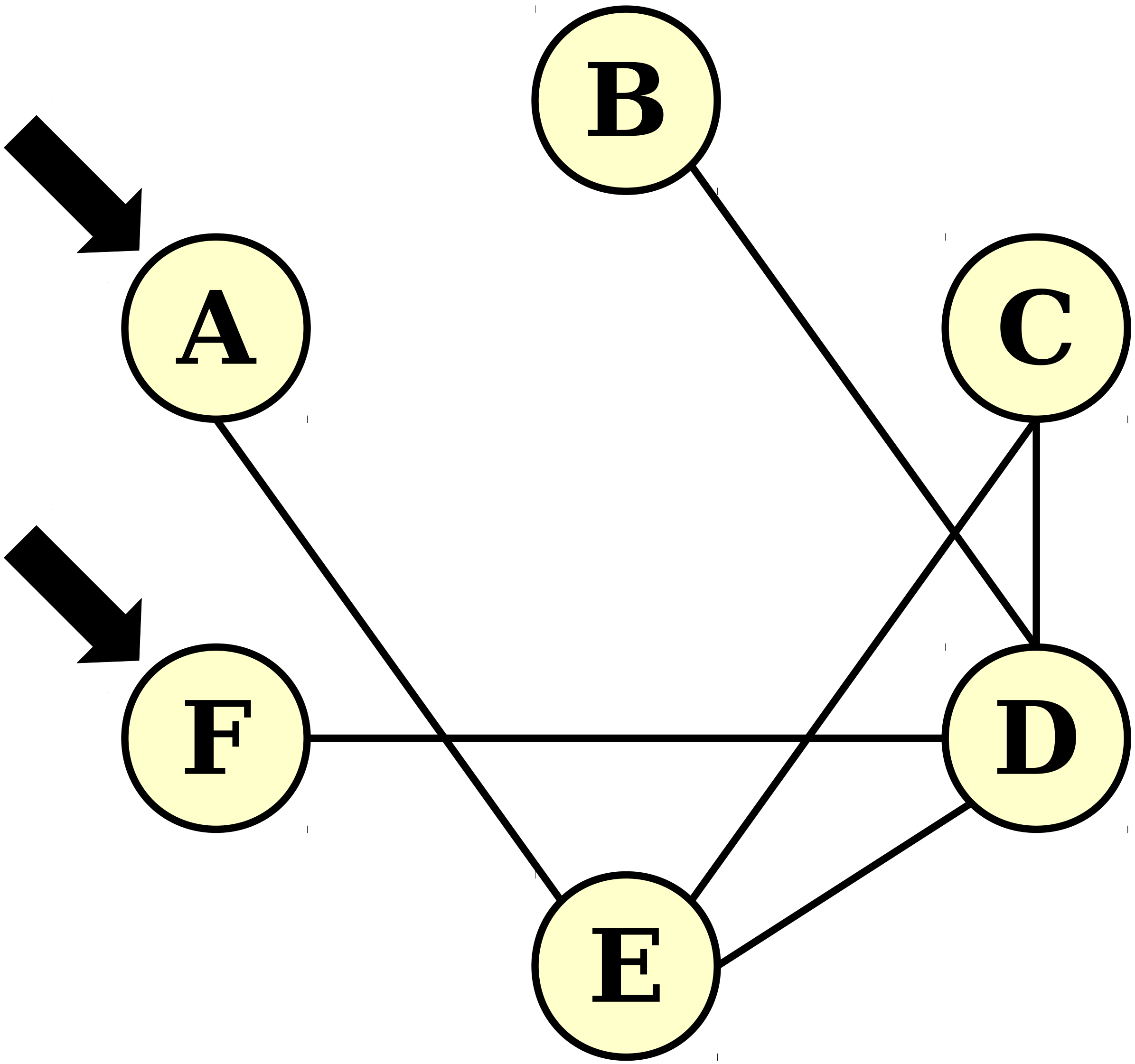
Another Problem



Another Problem



Another Problem



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

Shortest Paths

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

Shortest Paths

- ***Theorem:*** It's possible to find the shortest path between two nodes in an n-node, m-edge graph in time $O(m + n)$.
- ***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

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

$4n^2 - 3n + 137$

10^{500}

2^n

1.000000000001^n

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

$n^{\log n}$

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

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**

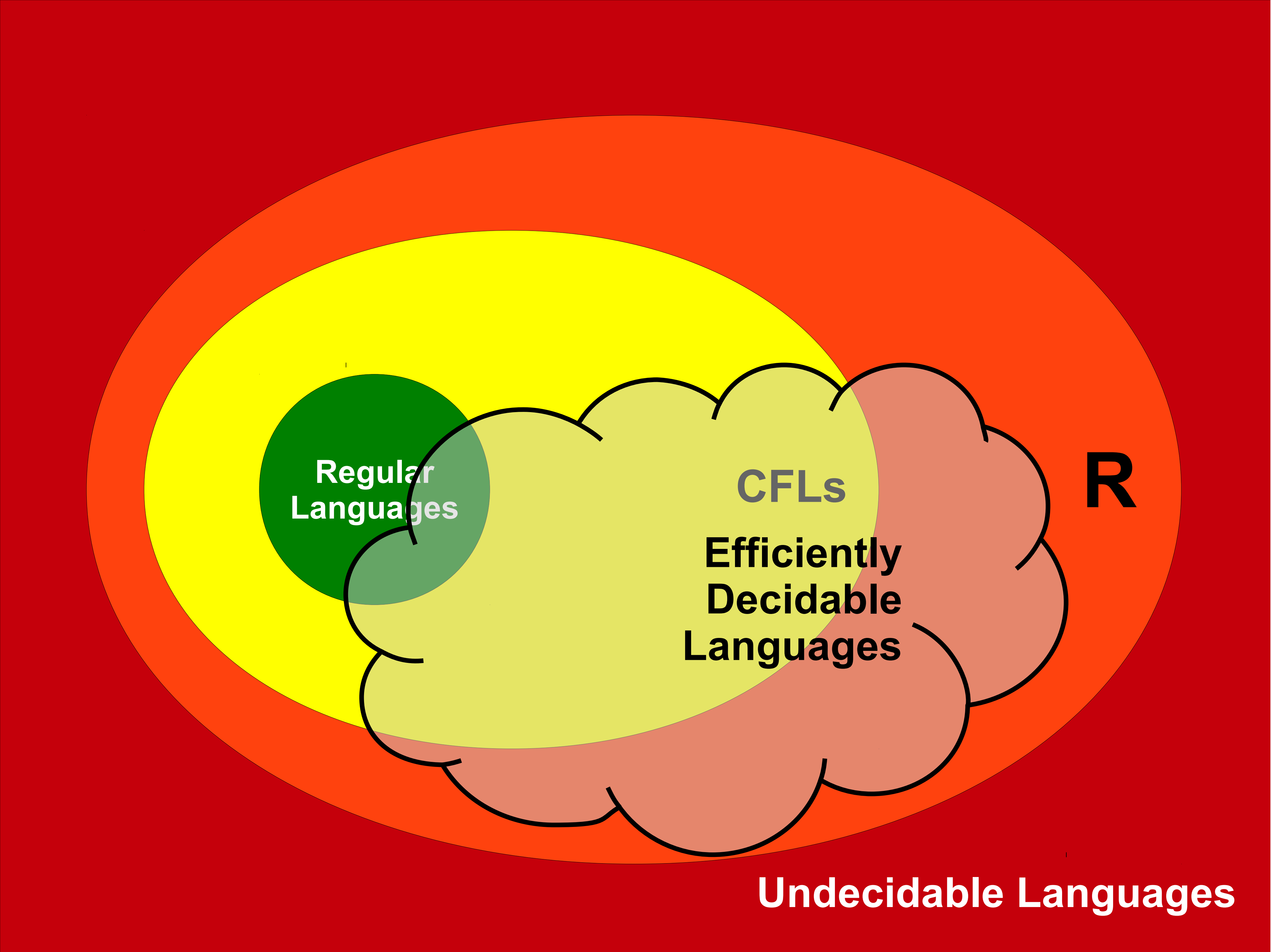
- The ***complexity class*** **P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

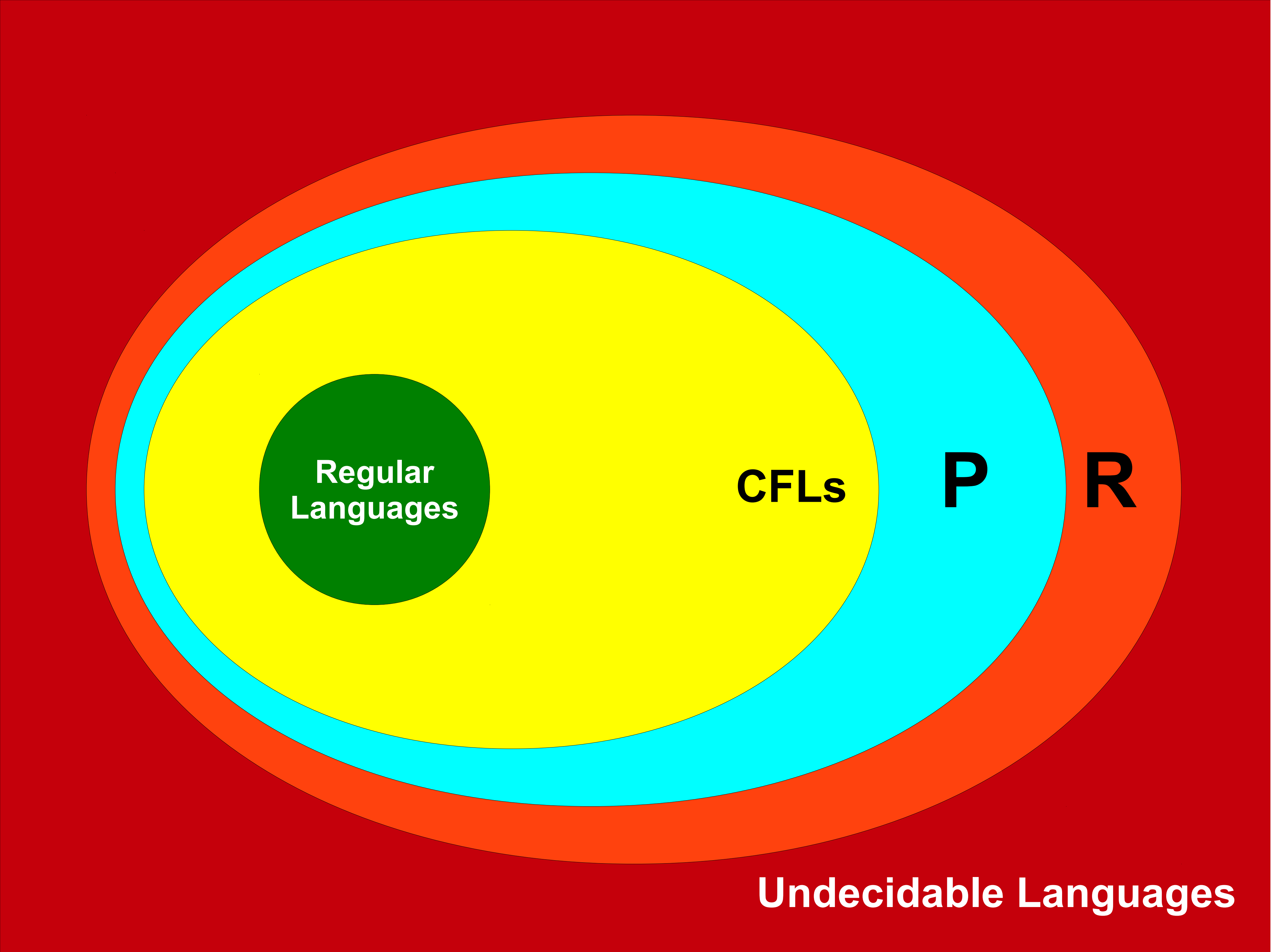
  P = { L | There is a polynomial-time decider for L }

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

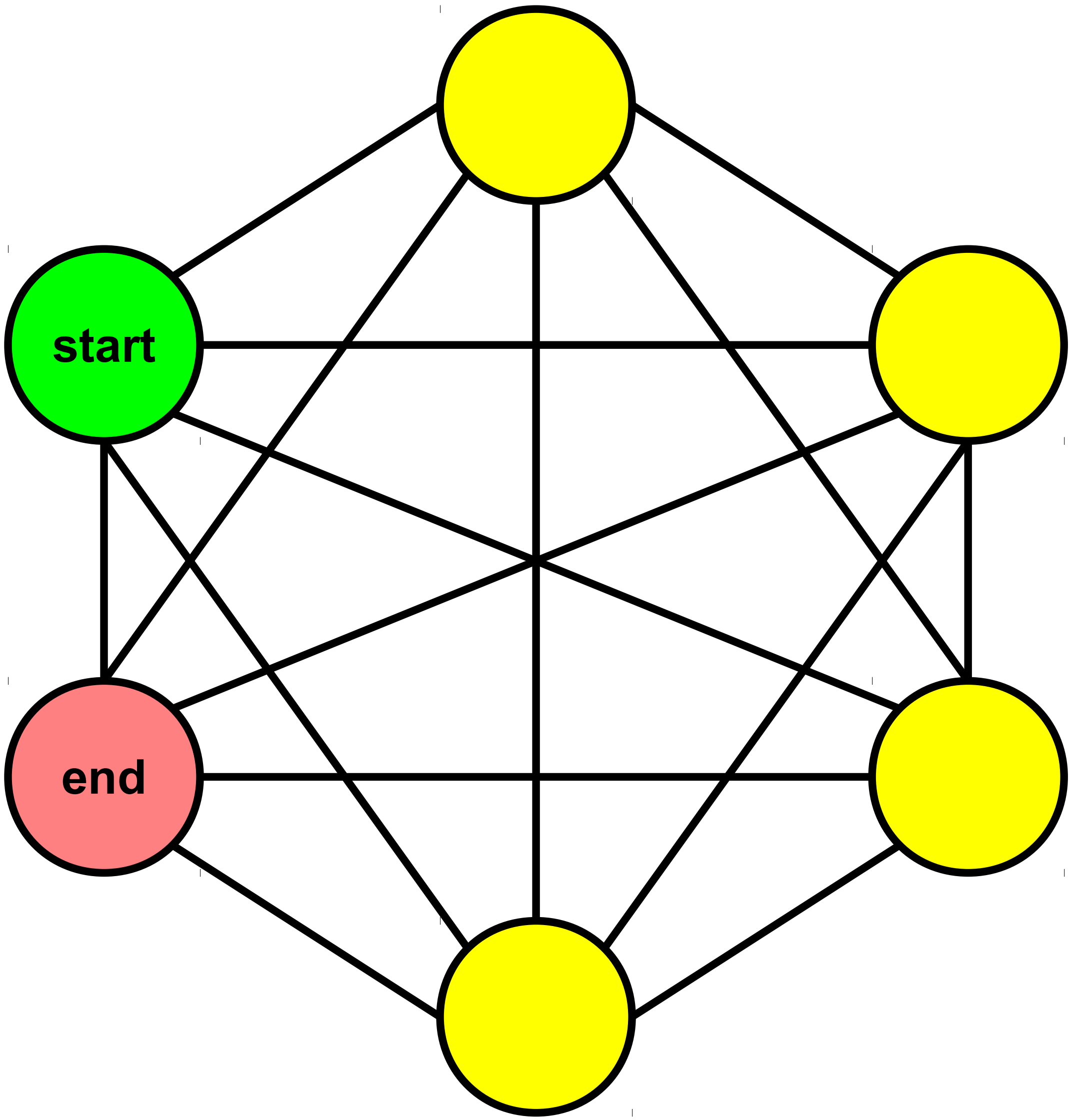
Examples of Problems in **P**

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


Regular Languages
CFLs
Efficiently Decidable Languages
R
Undecidable Languages


Regular Languages
CFLs
P
R
Undecidable Languages

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



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

List all the subsets of a given set.

Calculate 2^n for a given n, where the input and output are both written in unary (base 1).

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

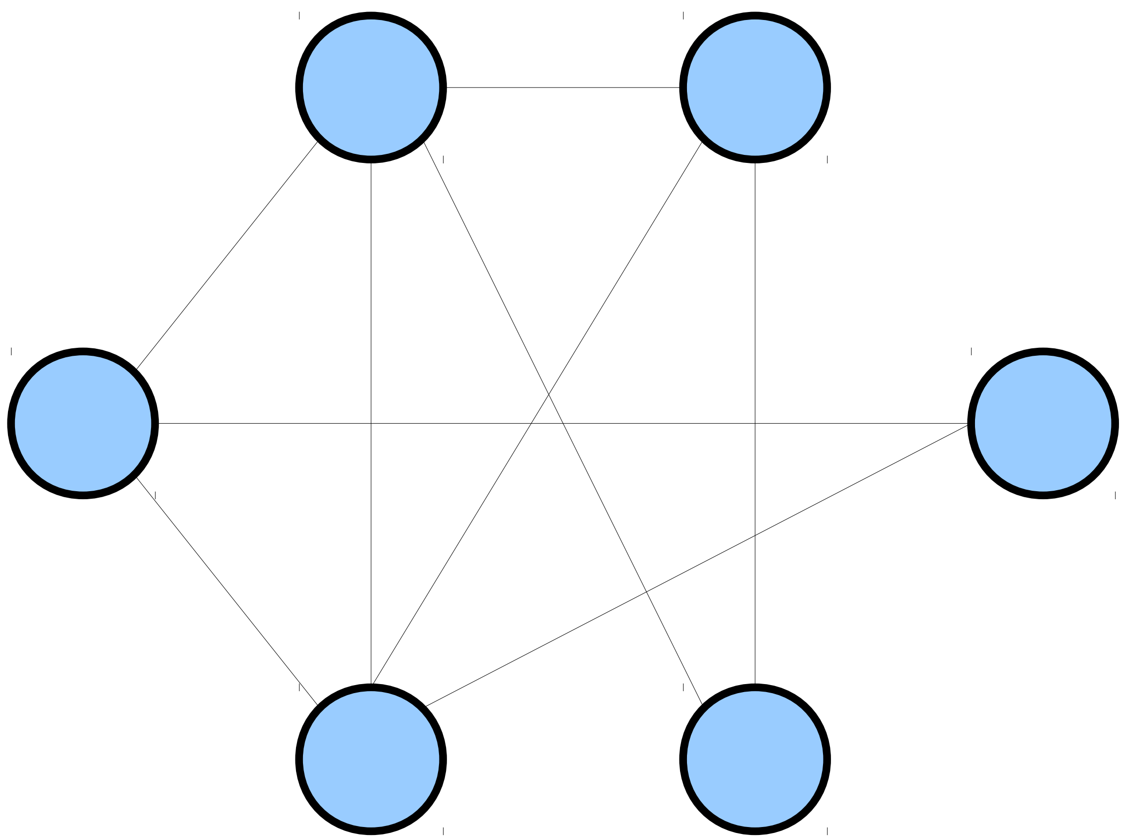
Is there an ascending subsequence of length at least 7?

Verifiers – Again

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

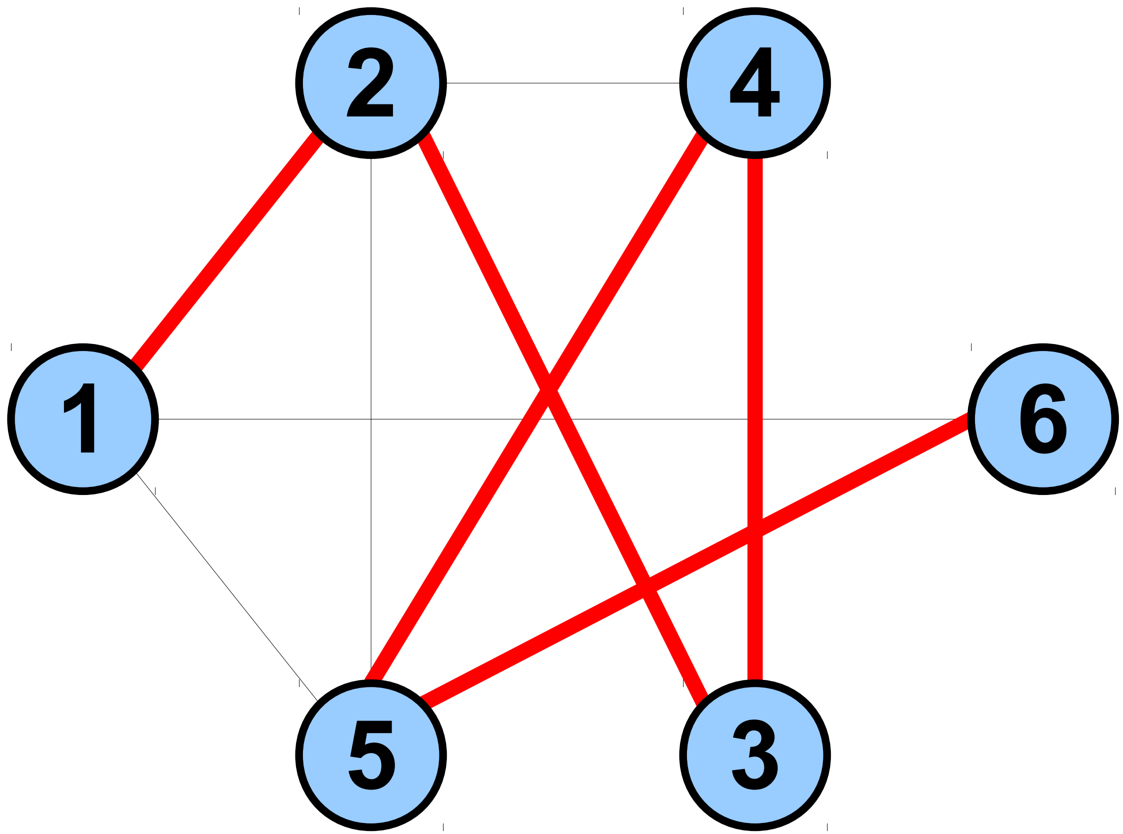
Is there an ascending subsequence of length at least 7?

Verifiers – Again



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

Verifiers – Again



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

Verifiers

- Recall that a ***verifier*** for L is a TM V such that
  - V halts on all inputs.
  - $w \in L \quad \text{iff} \quad \exists c \in \Sigma^*.\ V$ 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.
  - $w \in L \quad \text{iff} \quad \exists c \in \Sigma^*.\ V$ 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:

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

- The name **NP** comes from another way of characterizing **NP**. If you introduce *nondeterministic Turing machines* and appropriately define "polynomial time," then **NP** is the set of problems that an NTM can solve in polynomial time.
- Although it's not immediately obvious, **NP** $\subsetneq$ **R**. Come talk to me after class if you're curious why!

And now...

The

Most Important Question

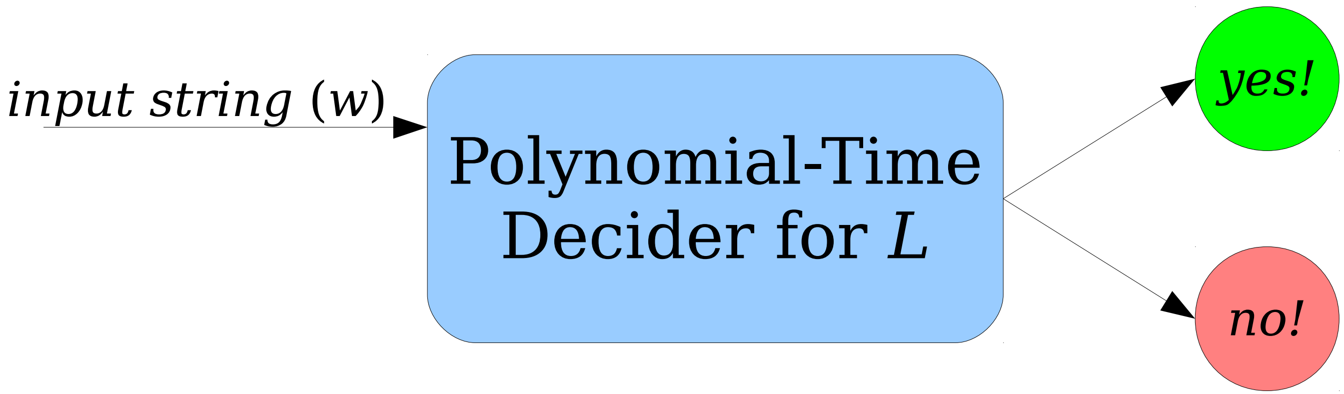
in

Theoretical Computer Science

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

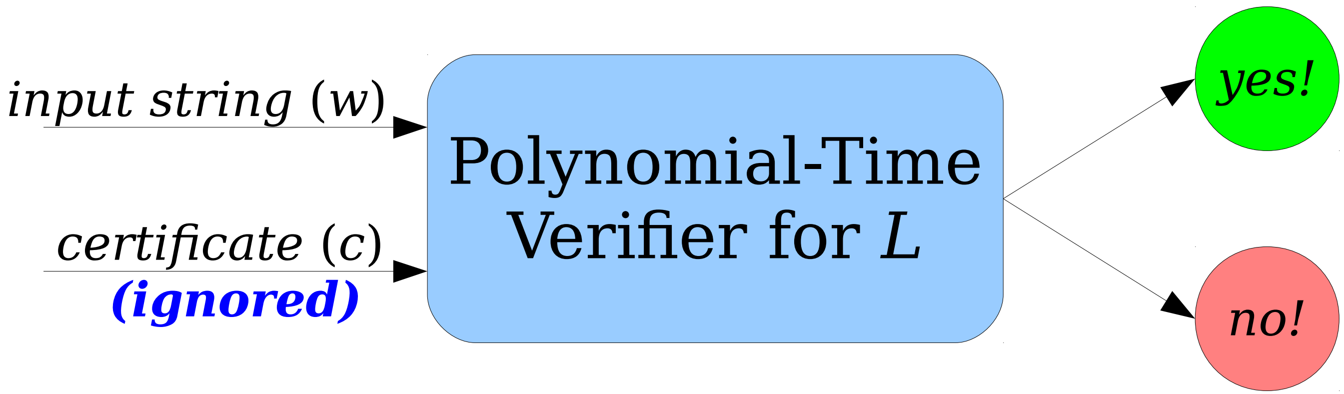
P = { L | There is a polynomial-time decider for L }

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



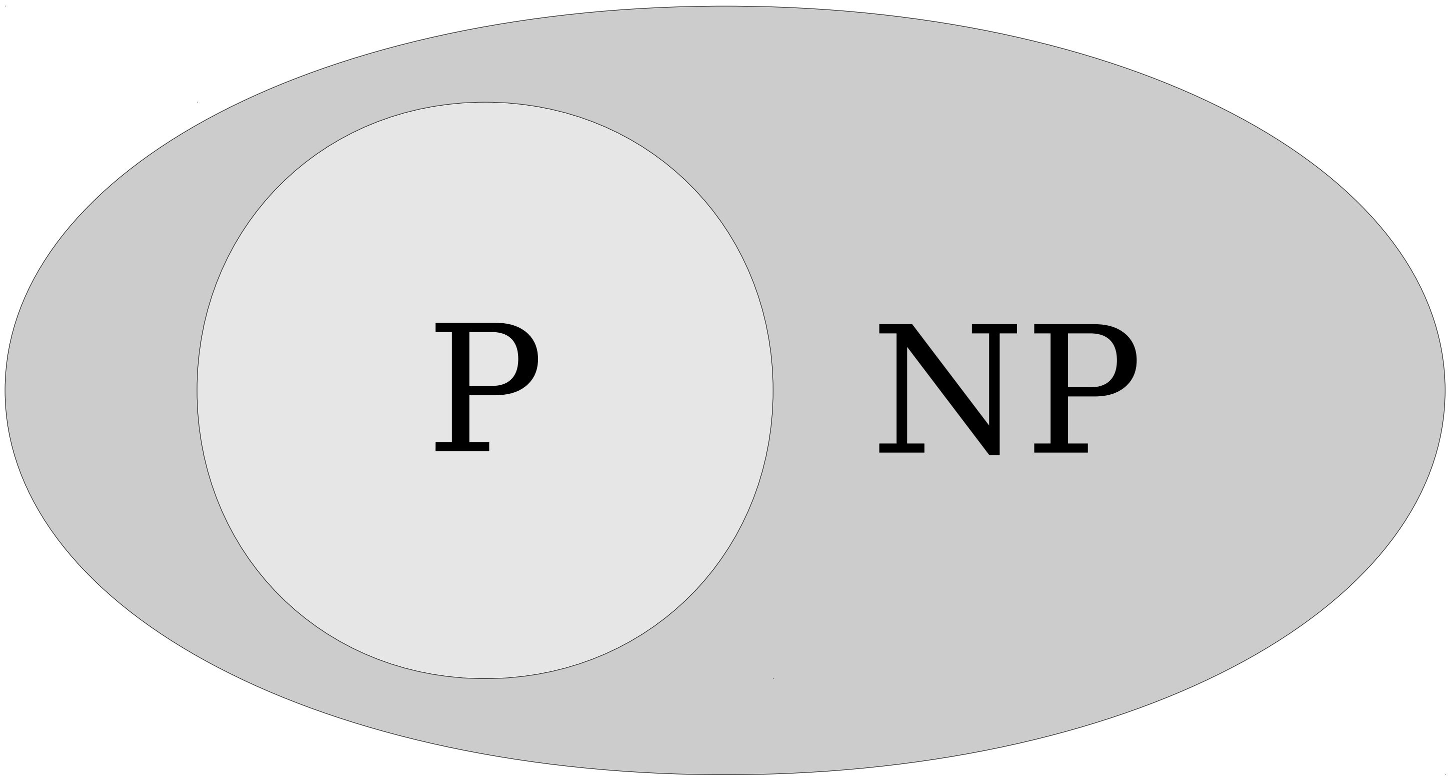
P = { L | There is a polynomial-time decider for L }

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

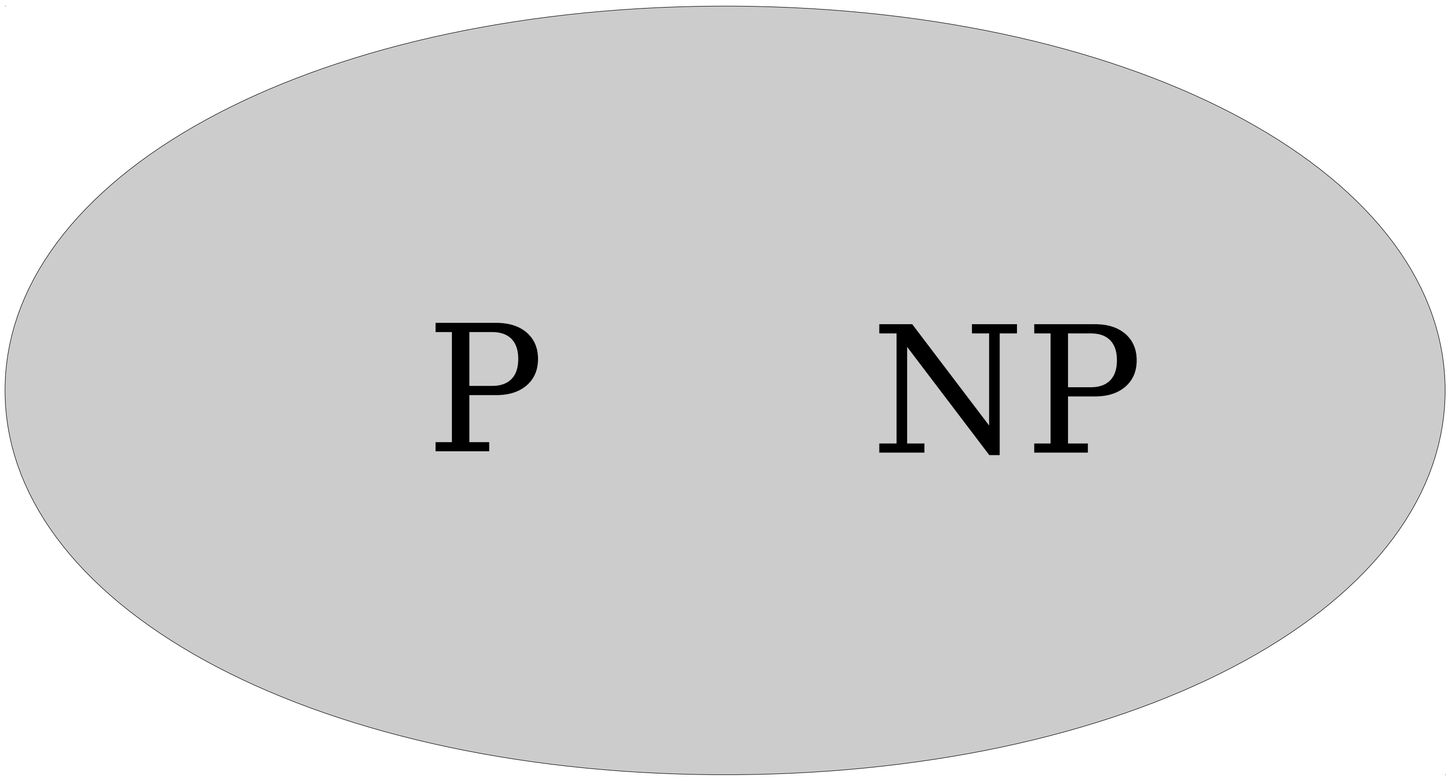


$$\mathbf{P} \subseteq \mathbf{NP}$$

Which Picture is Correct?



Which Picture is Correct?



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

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

- The $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question is the most important question in theoretical computer science.
- With the verifier definition of **NP**, one way of phrasing this question is

  *If a solution to a problem can be **checked** efficiently, can that problem be **solved** efficiently?*

- An answer either way will give fundamental insights into the nature of computation.

Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem).
  - Determining whether a simple DNA strand exists that multiple gene sequences could be a part of (shortest common supersequence).
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation).
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling).
  - ***And many more***.
- If **P = NP**, ***all*** of these problems have efficient solutions.
- If **P ≠ NP**, ***none*** of these problems have efficient solutions.

Why This Matters

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

What We Know

- Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ has proven ***extremely difficult***.
- In the past 45 years:
  - Not a single correct proof either way has been found.
  - Many types of proofs have been shown to be insufficiently powerful to determine whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.
  - A majority of computer scientists believe $\mathbf{P} \neq \mathbf{NP}$, but this isn't a large majority.
- Interesting read: Interviews with leading thinkers about $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$:
  - http://web.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**.

Do you think **P = NP**?

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **Y** or **N**.

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

Adapting our Techniques

A Problem

- The **R** and **RE** languages correspond to problems that can be decided and verified, *period*, without any time bounds.
- To reason about what's in **R** and what's in **RE**, we used two key techniques:
  - ***Universality***: TMs can run other TMs as subroutines.
  - ***Self-Reference***: TMs can get their own source code.
- Why can't we just do that for **P** and **NP**?

Theorem (Baker-Gill-Solovay): Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Proof: Take CS154!

So how *are* we going to reason about **P** and **NP**?

Next Time

- ***Reducibility***
  - A technique for connecting problems to one another.
- ***NP-Completeness***
  - What are the hardest problems in **NP**?