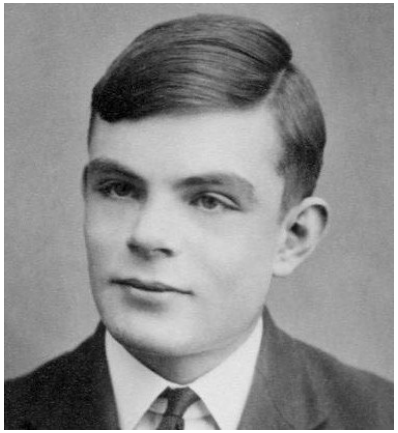
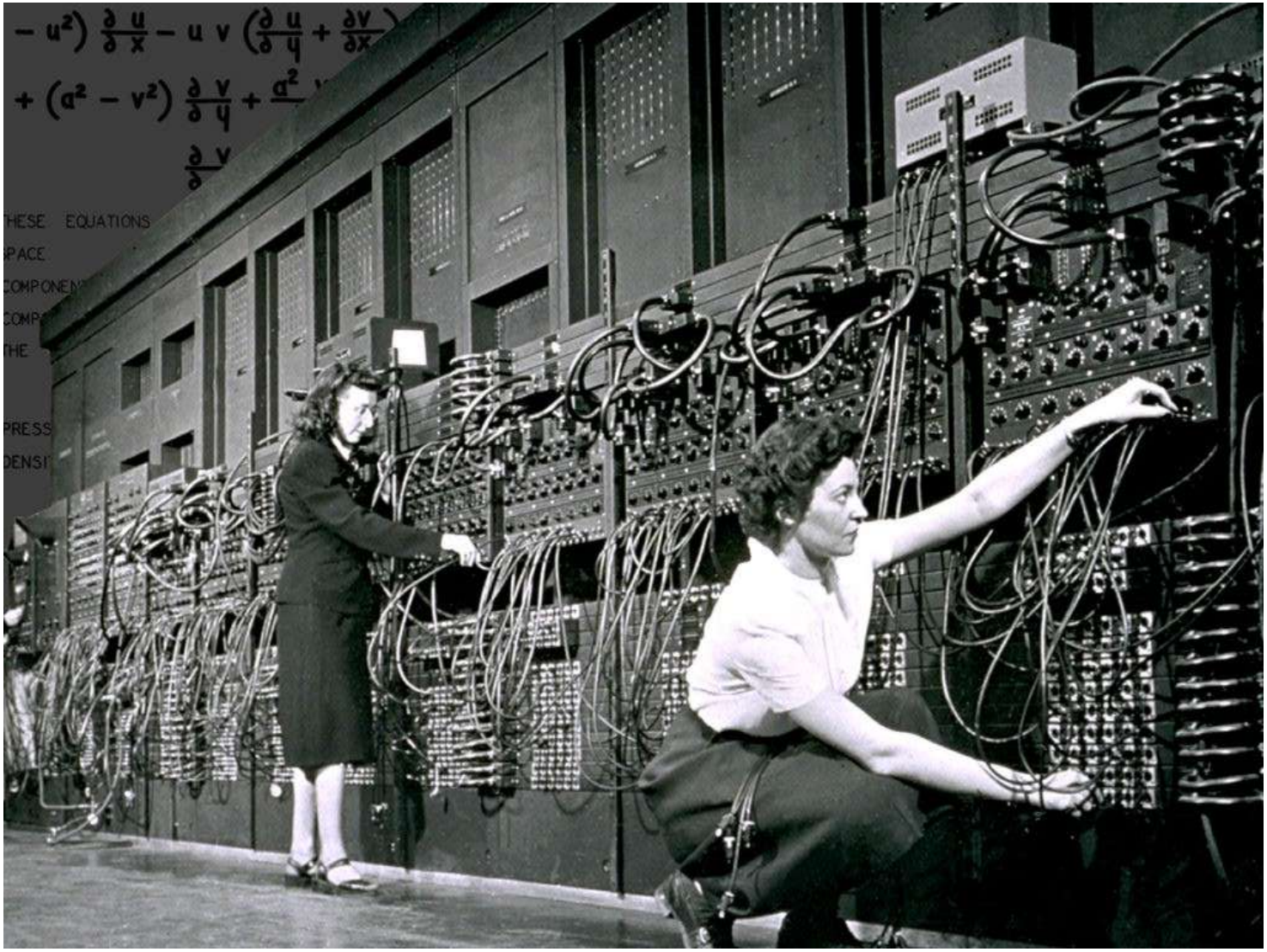
Complexity Theory



Alan Turing
On Computable Numbers
1936



ENIAC
First General-Purpose Electronic Digital Computer
1945

A Decidable Problem

- Consider the following problem:

  Given two regular expressions R_1 and R_2, determine whether R_1 and R_2 have the same language.

- This problem is indeed decidable.
  - We autograded your regular expressions in Problem Set Seven. The algorithm we used is 100% accurate.
- ***Theorem:*** There is no algorithm for solving this problem whose runtime is $O(2^{m+n})$, where m and n are the lengths of the input regular expressions.

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?

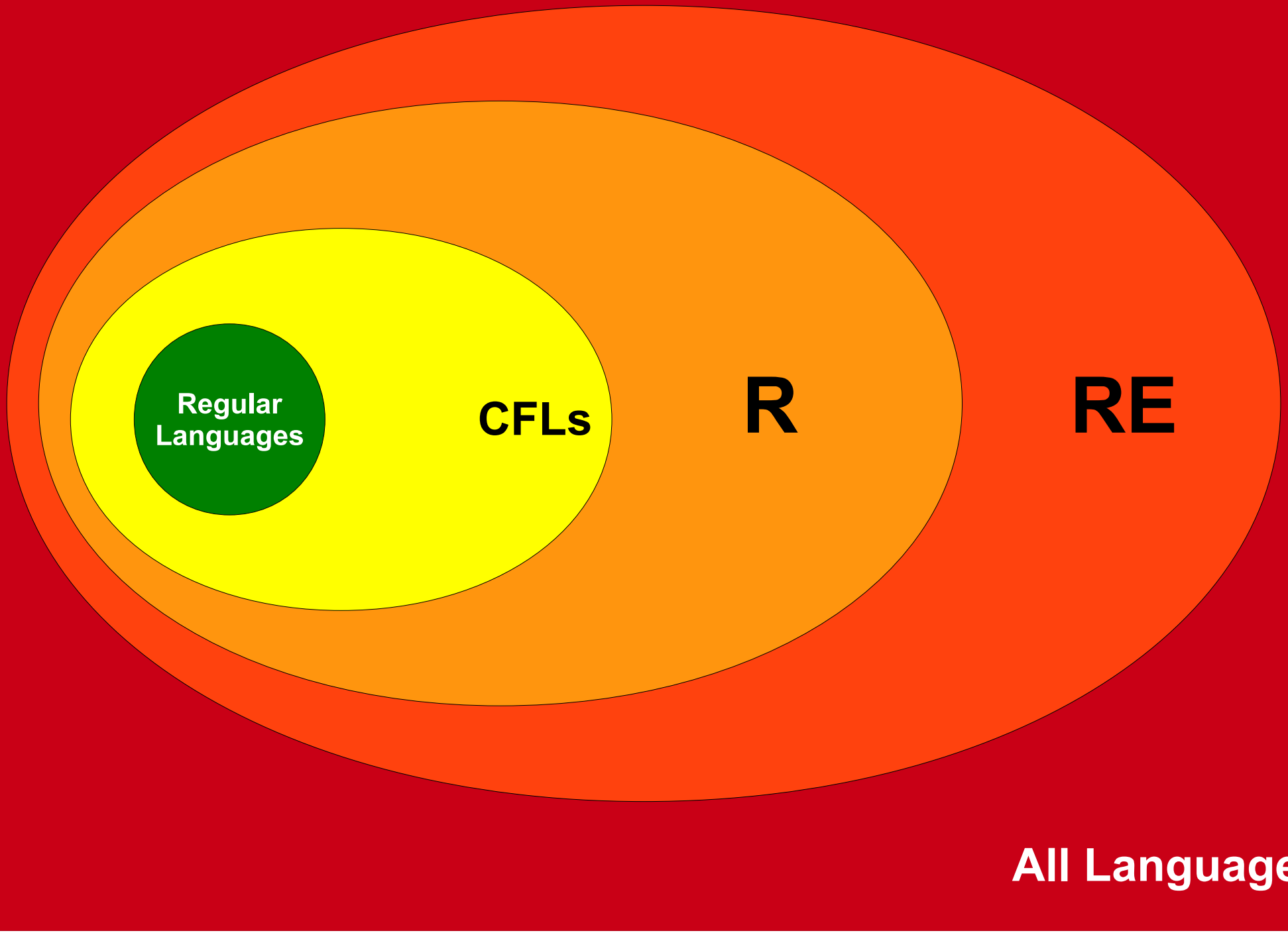
- In the remainder of this course, we will explore this question in more detail.

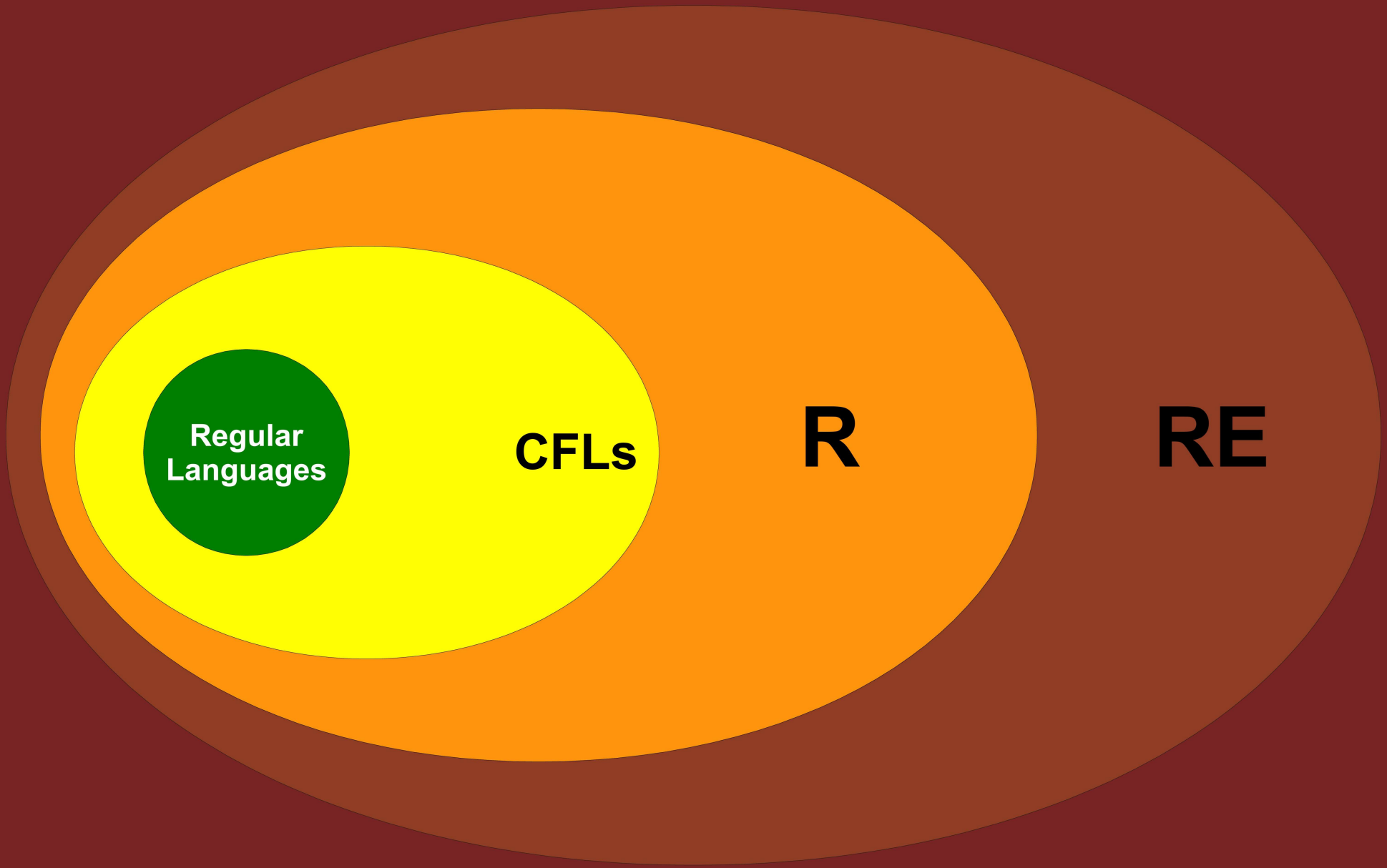
Where We've Been

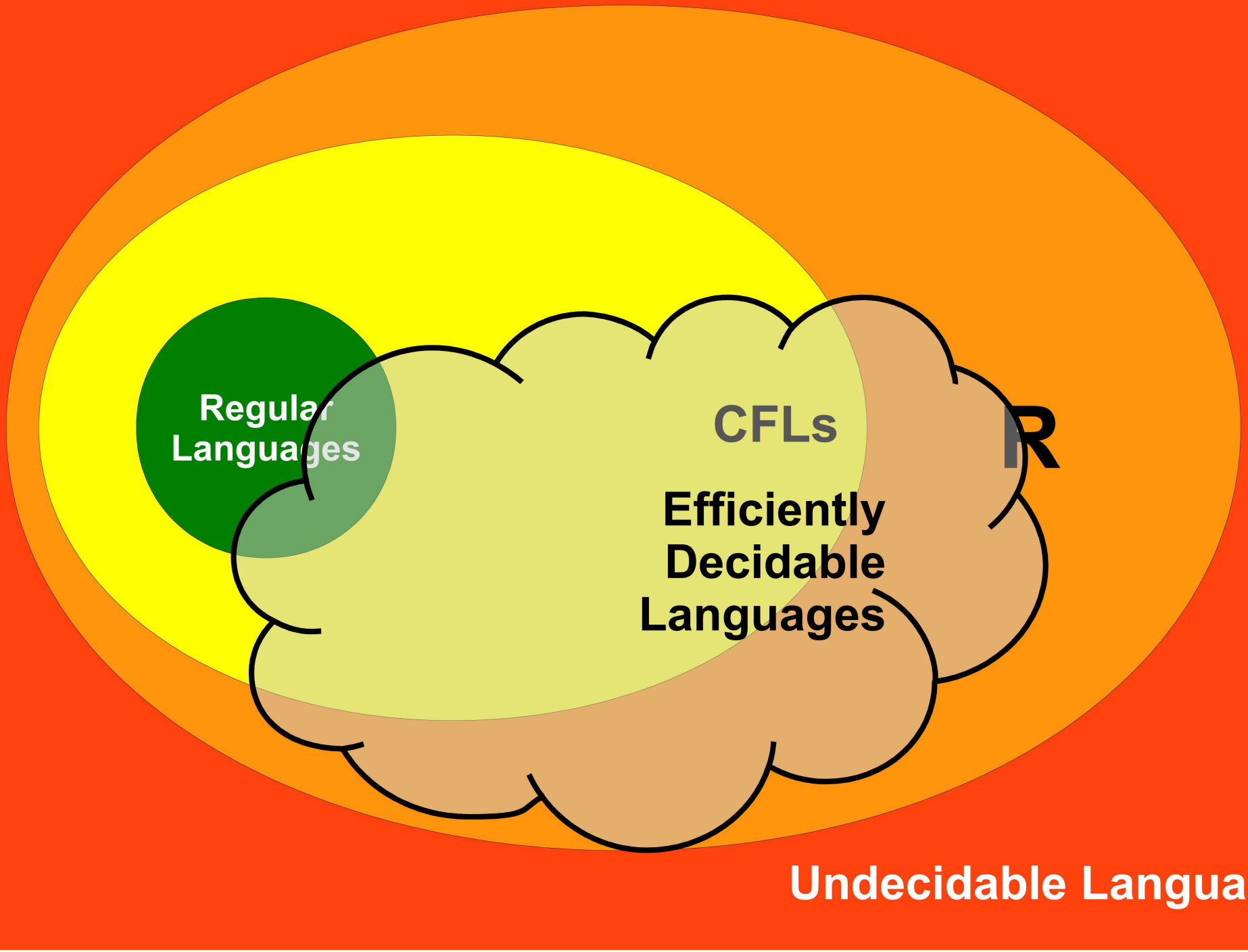
- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.

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 resources do we want our programs to make “efficient” use of?
  - How do we draw the line between “efficient” and “inefficient?”

Correctness

- We have a program written in your Favorite Programming Language that's a decider for some problem.
- What does it mean for this program to be correct?

Go to PollEv.com/cs103spr25

Efficiency

- We have a program written in your Favorite Programming Language that's a decider for some problem.
- What aspect of that program might quantify "efficiency?"
  - The number of lines of code in the program.
  - How deeply-nested the loops or recursion in the program are.
  - How much time it takes for the program to solve the problem.
  - How much memory it takes for the program to solve the problem.
  - How much power it takes for the program to solve the problem.
  - How much network communication it takes for the program to solve the problem.
  - …

Efficiency

- We have a program written in your Favorite Programming Language that's a decider for some problem.
- What aspect of that program might quantify "efficiency?"
  - The number of lines of code in the program.
  - How deeply-nested the loops or recursion in the program are.
  - How much time it takes for the program to solve the problem.
  - How much memory it takes for the program to solve the problem.
  - How much power it takes for the program to solve the problem.
  - How much network communication it takes for the program to solve the problem.
  - …

What is an efficient algorithm?

Let’s explore some problems and solutions and see what we notice!

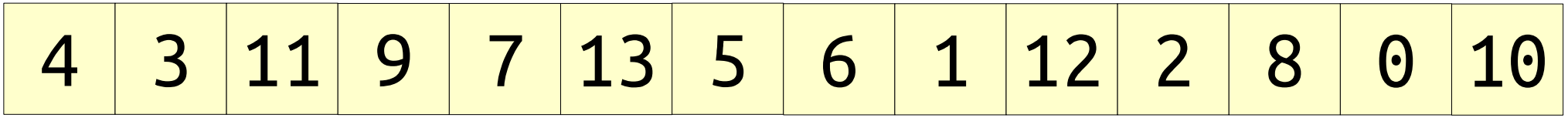
A Common Pattern: Searching Finite Spaces

- 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

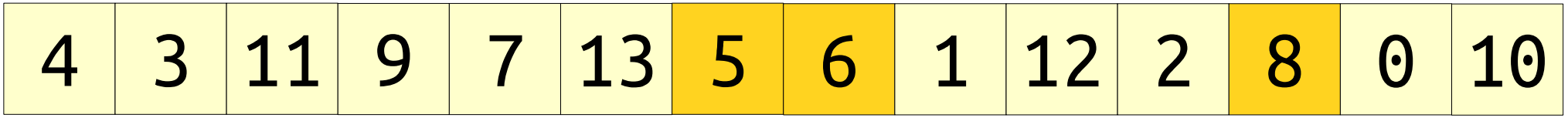
4	3	11	9	7	13	5	6	1	12	2	8	0	10

A Sample Problem



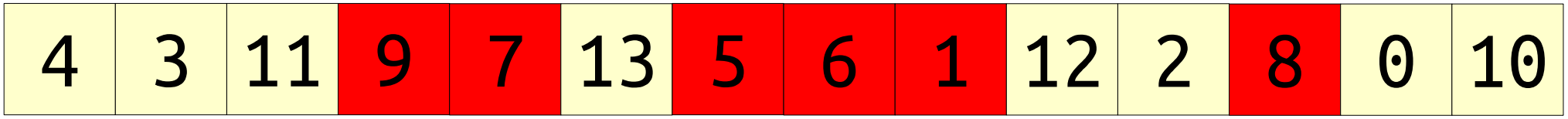
Goal: Find the length of the longest increasing subsequence of this sequence.

A Sample Problem



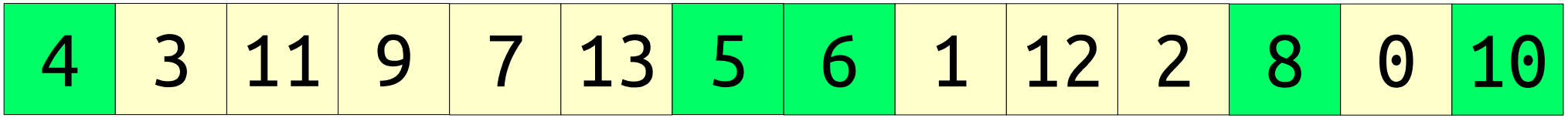
Goal: Find the length of the longest increasing subsequence of this sequence.

A Sample Problem



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

A Sample Problem



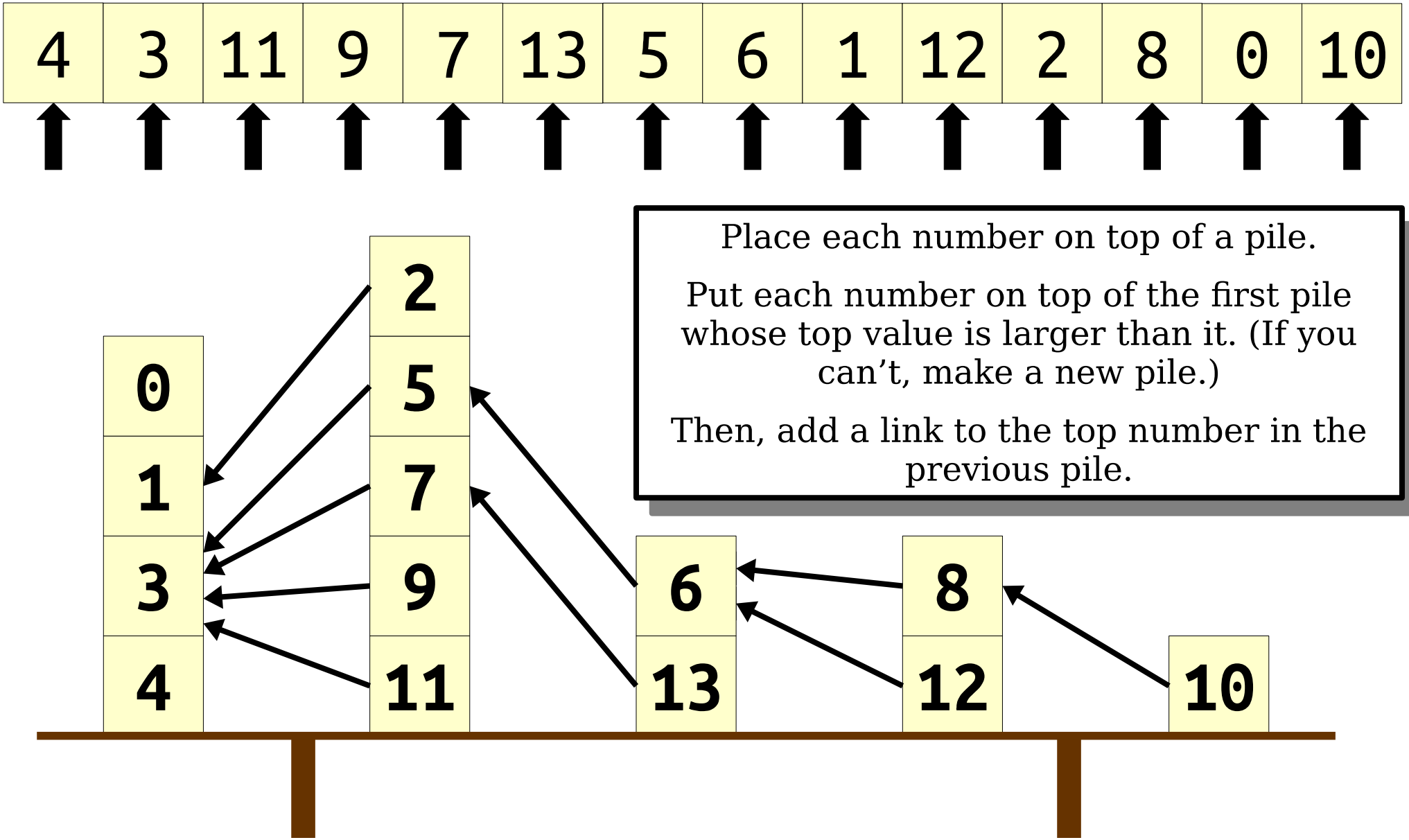
Goal: Find the length of the longest increasing subsequence of this sequence.

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 defines a subsequence.)
- Checking all of them to find the longest increasing subsequence will take time $O(n \cdot 2^n)$.
- ***Fact:*** the age of the universe is about 4.3×10^{26} nanoseconds. That's about 2^{85} nanoseconds.
- Practically speaking, this algorithm doesn't terminate if you give it an input of size 100 or more.

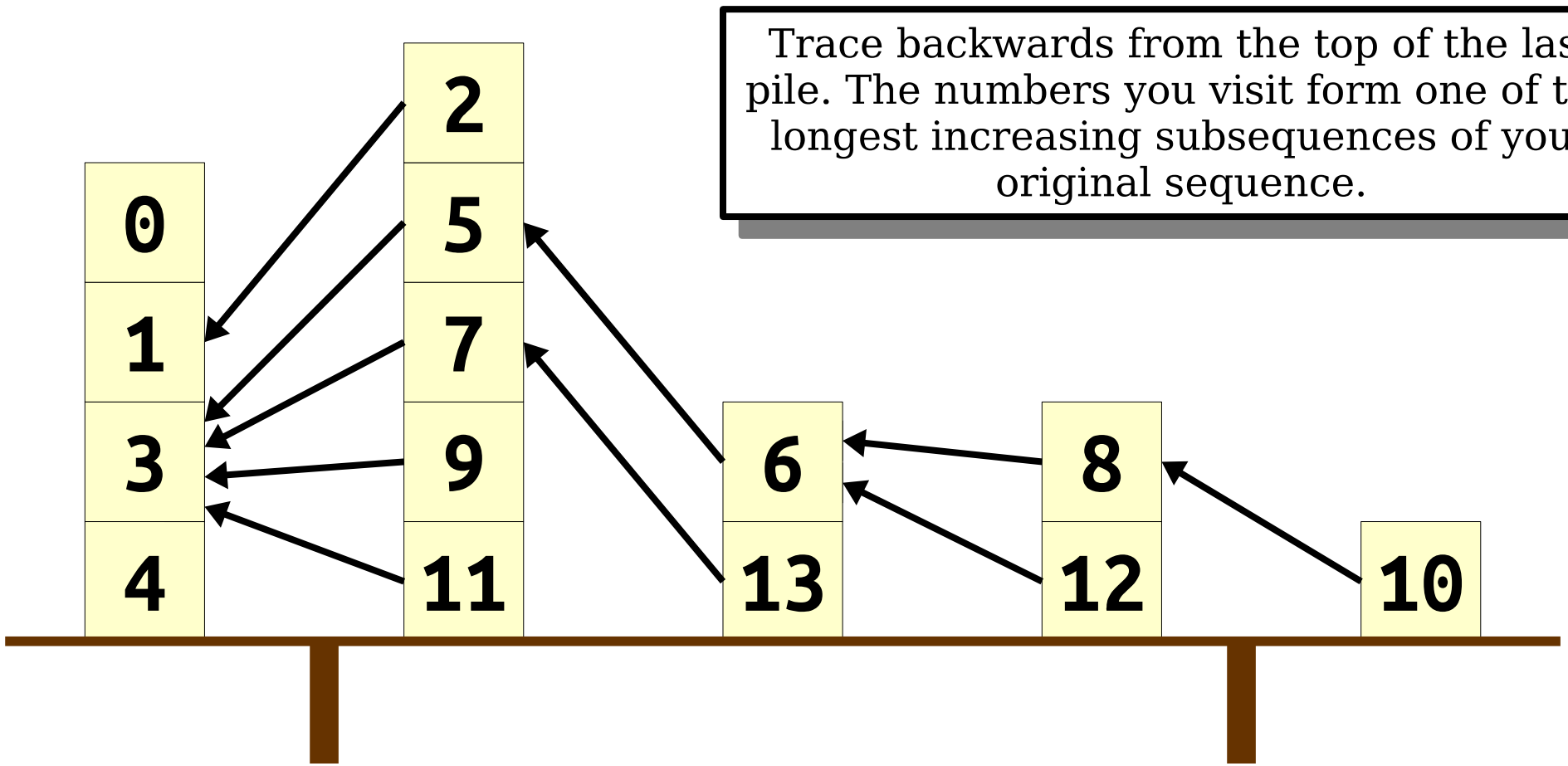
A Different Approach

Patience Sorting



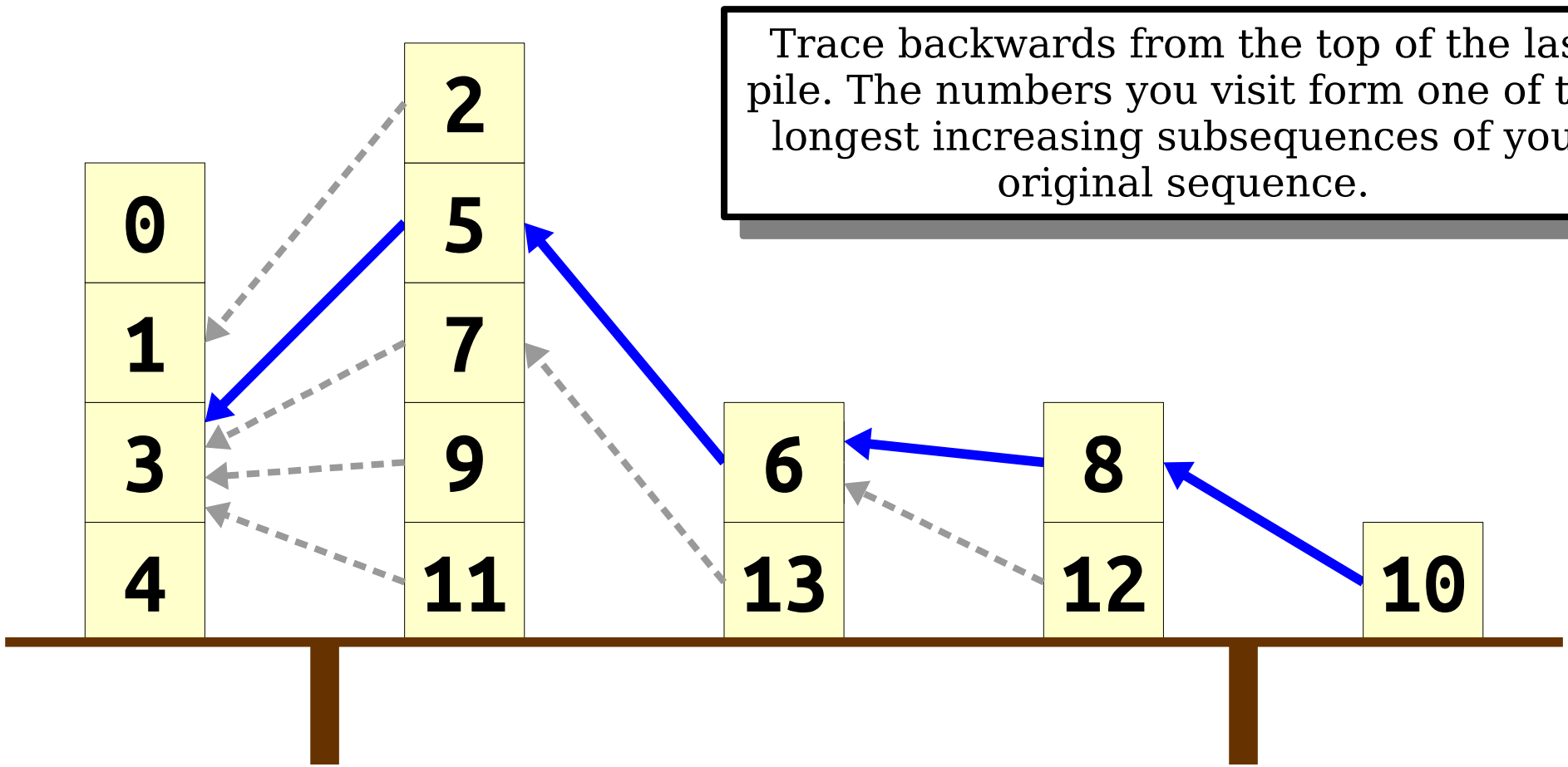
Patience Sorting

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



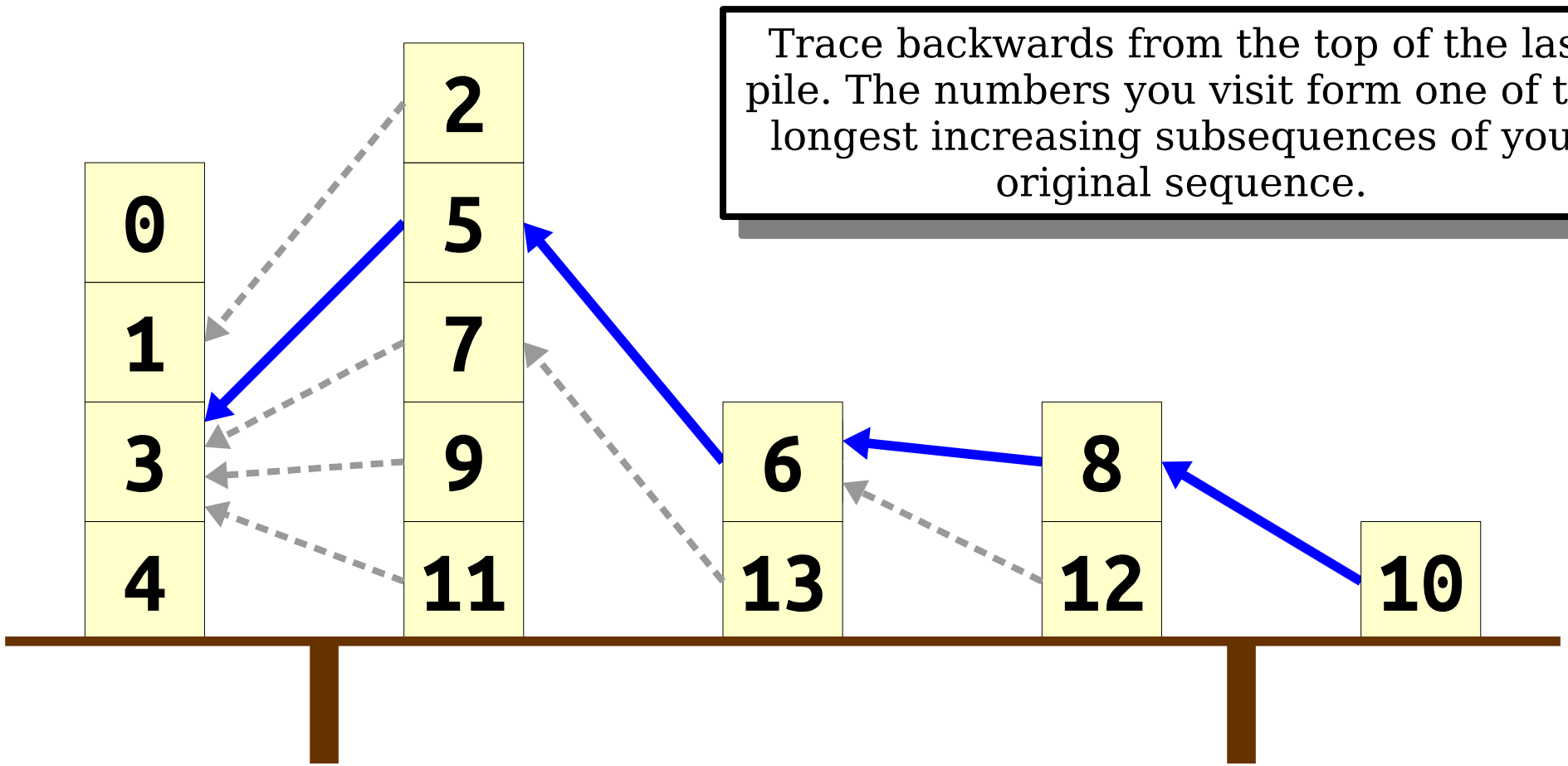
Patience Sorting

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



Patience Sorting

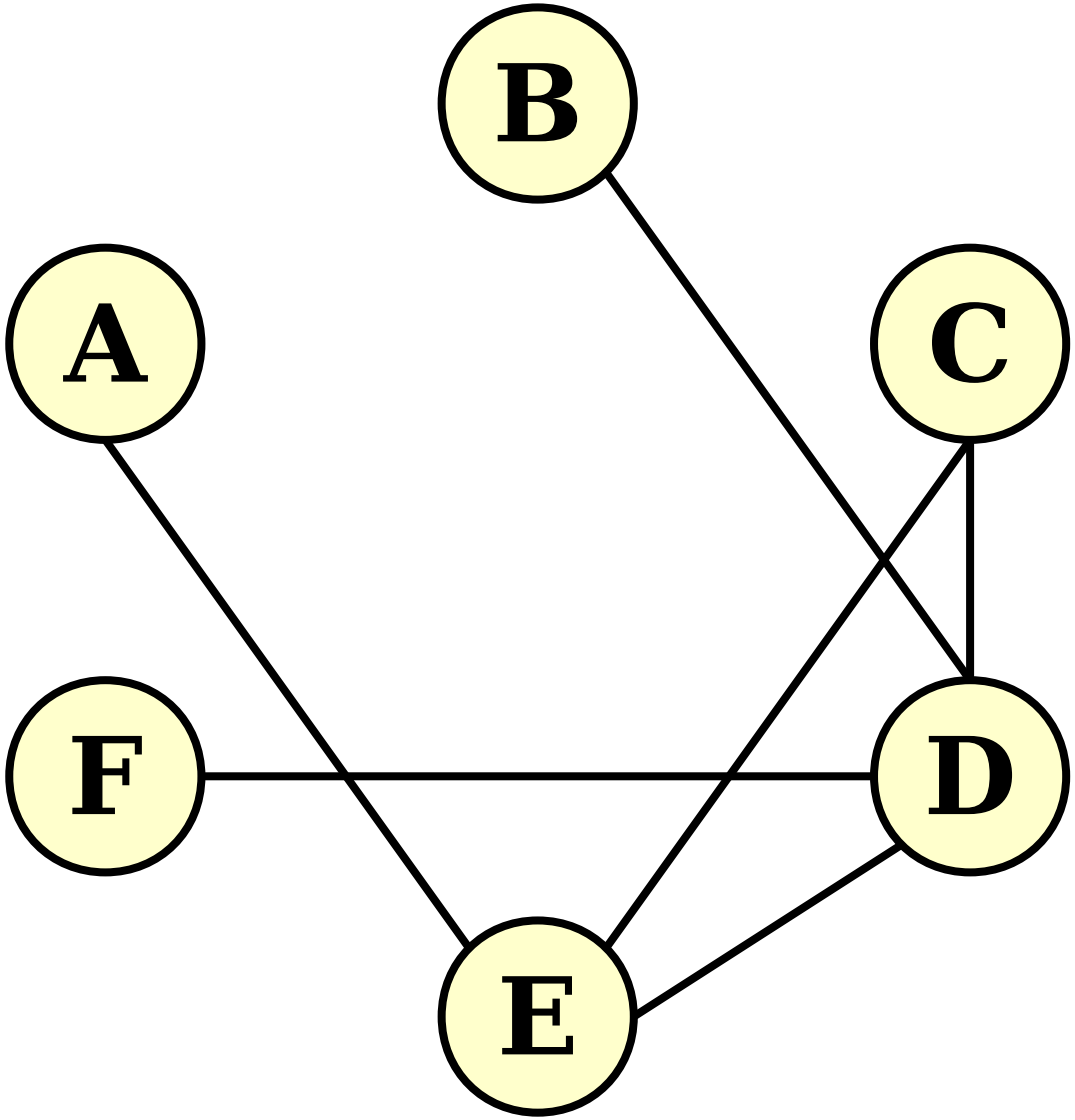
4	3	11	9	7	13	5	6	1	12	2	8	0	10



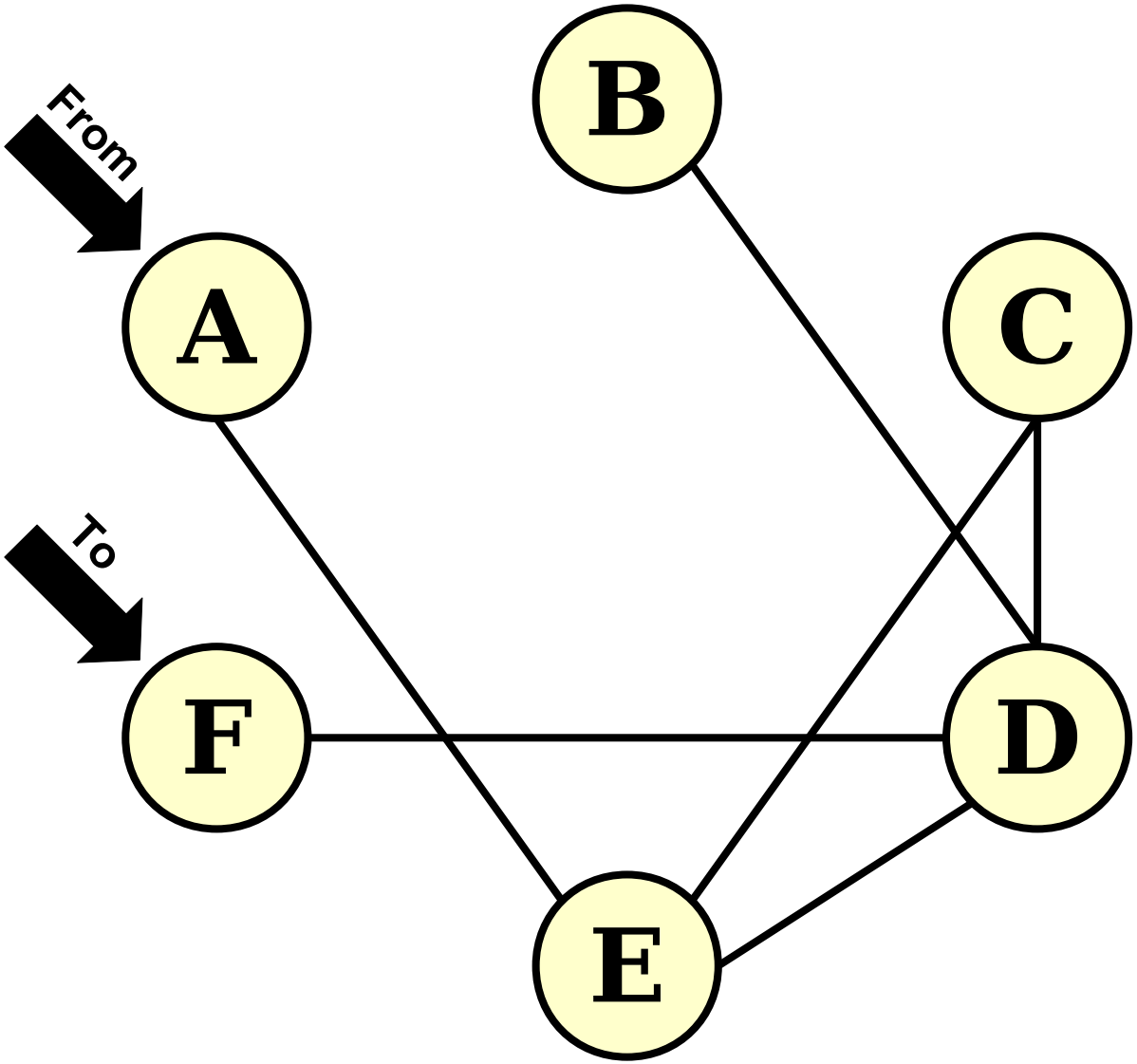
Longest Increasing Subsequences

- ***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.

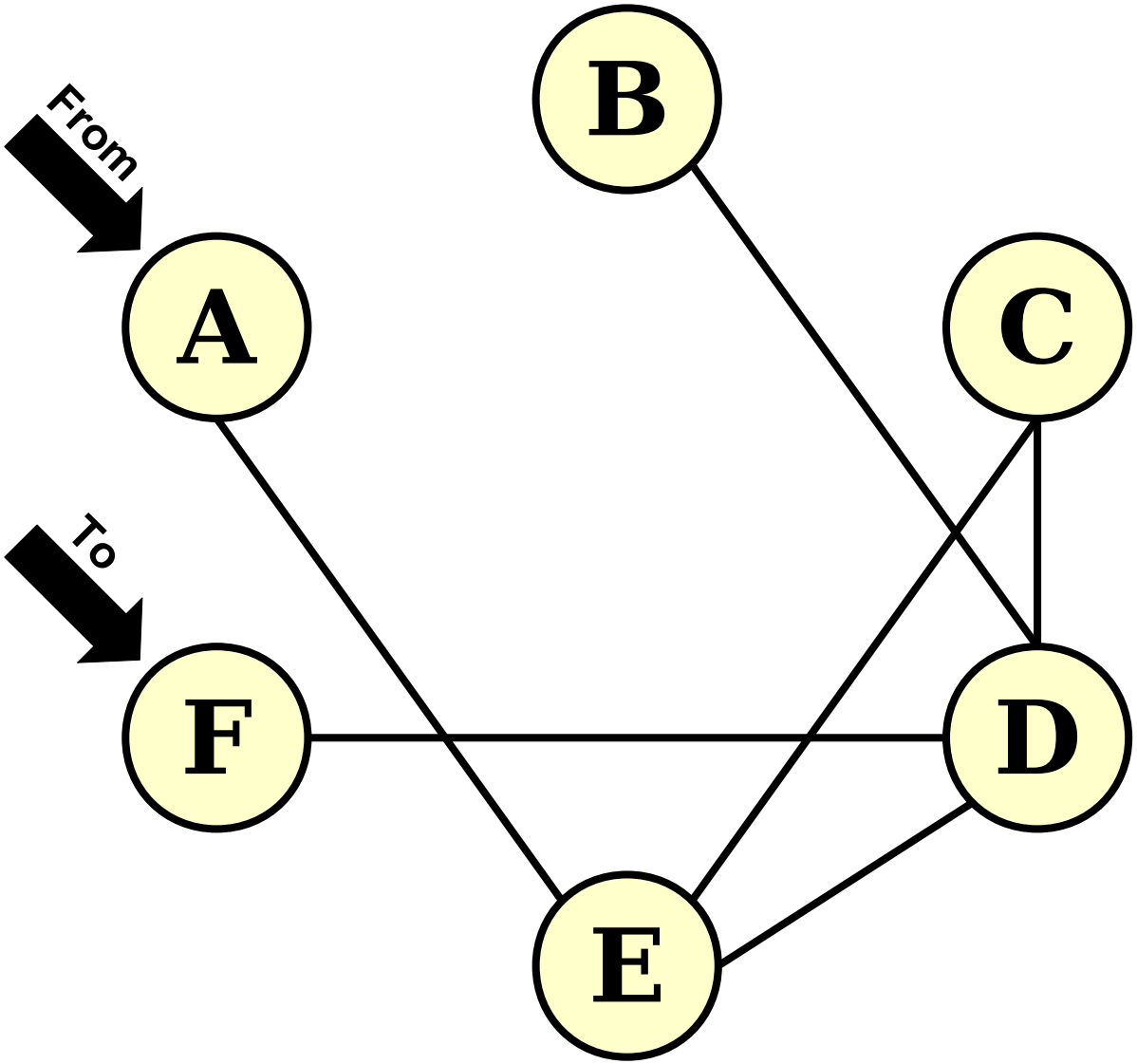
Another Problem



Another Problem



Another Problem



Goal: Determine the length of the shortest path from **F** to **A** 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

- **Pop Quiz!** How else could we find a shortest path?

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!
- This scales nicely!
- 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 in 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

Which of the following are considered efficient runtimes?

1	$n^2 - 3n + 17$
2	$n \log n$
3	$n^{1,000,000,000}$
4	n^n
5	$n!$
6	2^n
7	1.0000001^n
8	10^{500}

Go to
PollEv.com/cs103spr25

The Cobham-Edmonds Thesis

Which of the following are considered efficient runtimes?

1	$n^2 - 3n + 17$	✓	This is a polynomial in n.
2	$n \log n$	✓	Bounded by n^2.
3	$n^{1,000,000,000}$	✓	This is a polynomial in n.
4	n^n	✗	Eventually bigger than n^k for all k.
5	$n!$	✗	Eventually bigger than n^k for all k.
6	2^n	✗	Eventually bigger than n^k for all k.
7	1.0000001^n	✗	Eventually bigger than n^k for all k.
8	10^{500}	✓	$10^{500} = 10^{500}\ n^0$ is a polynomial in 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**

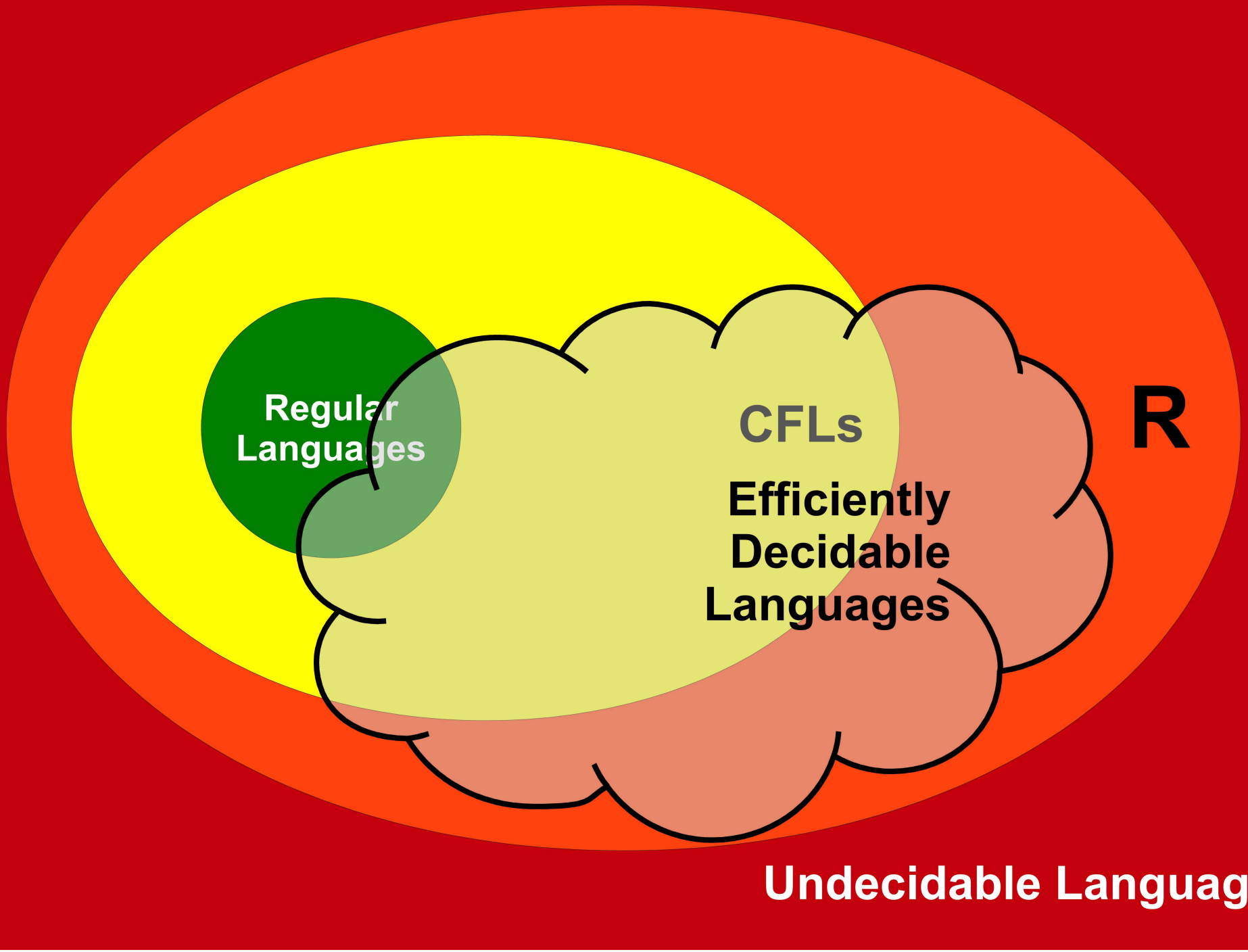
- The ***complexity class* P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

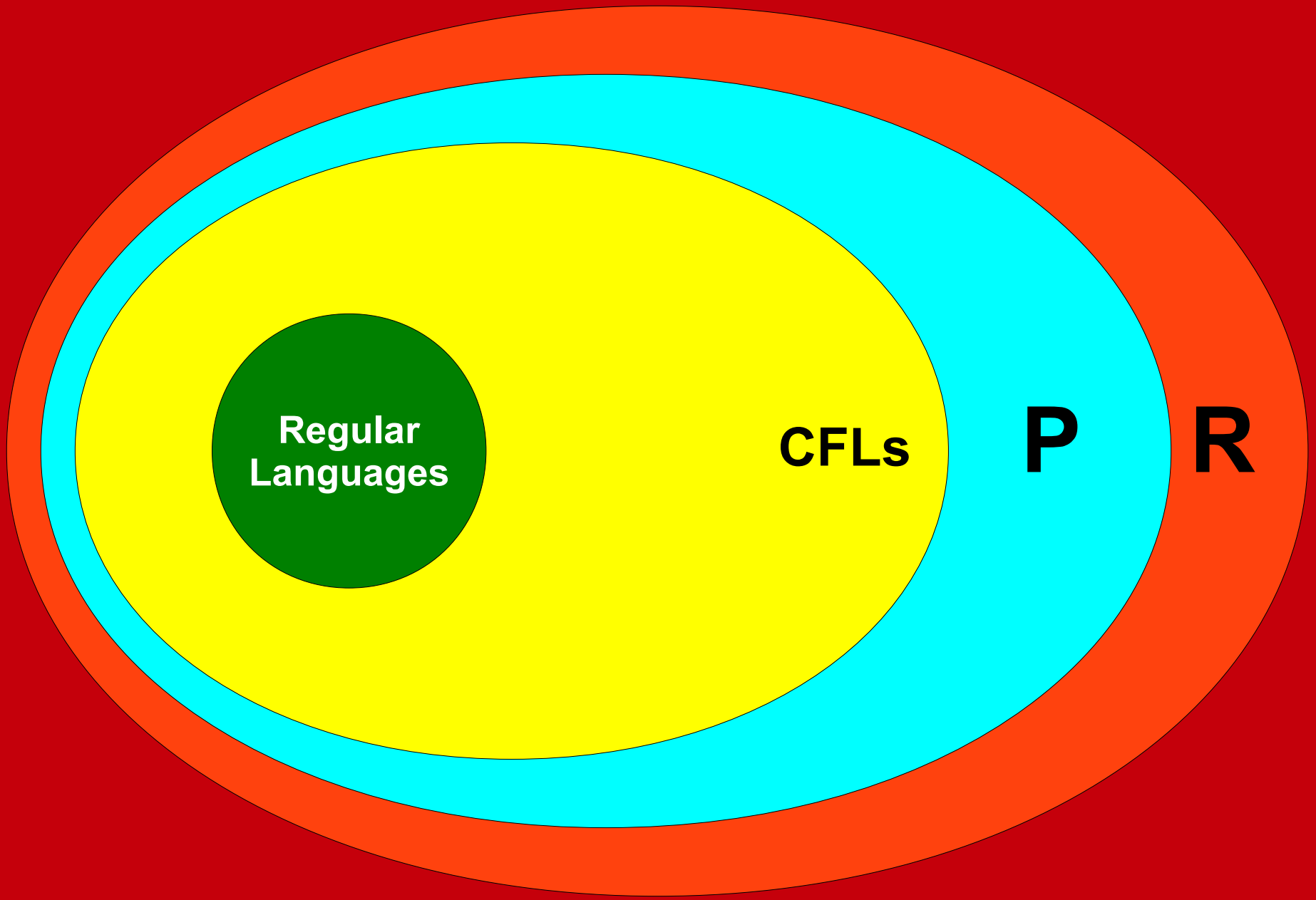
$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$

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

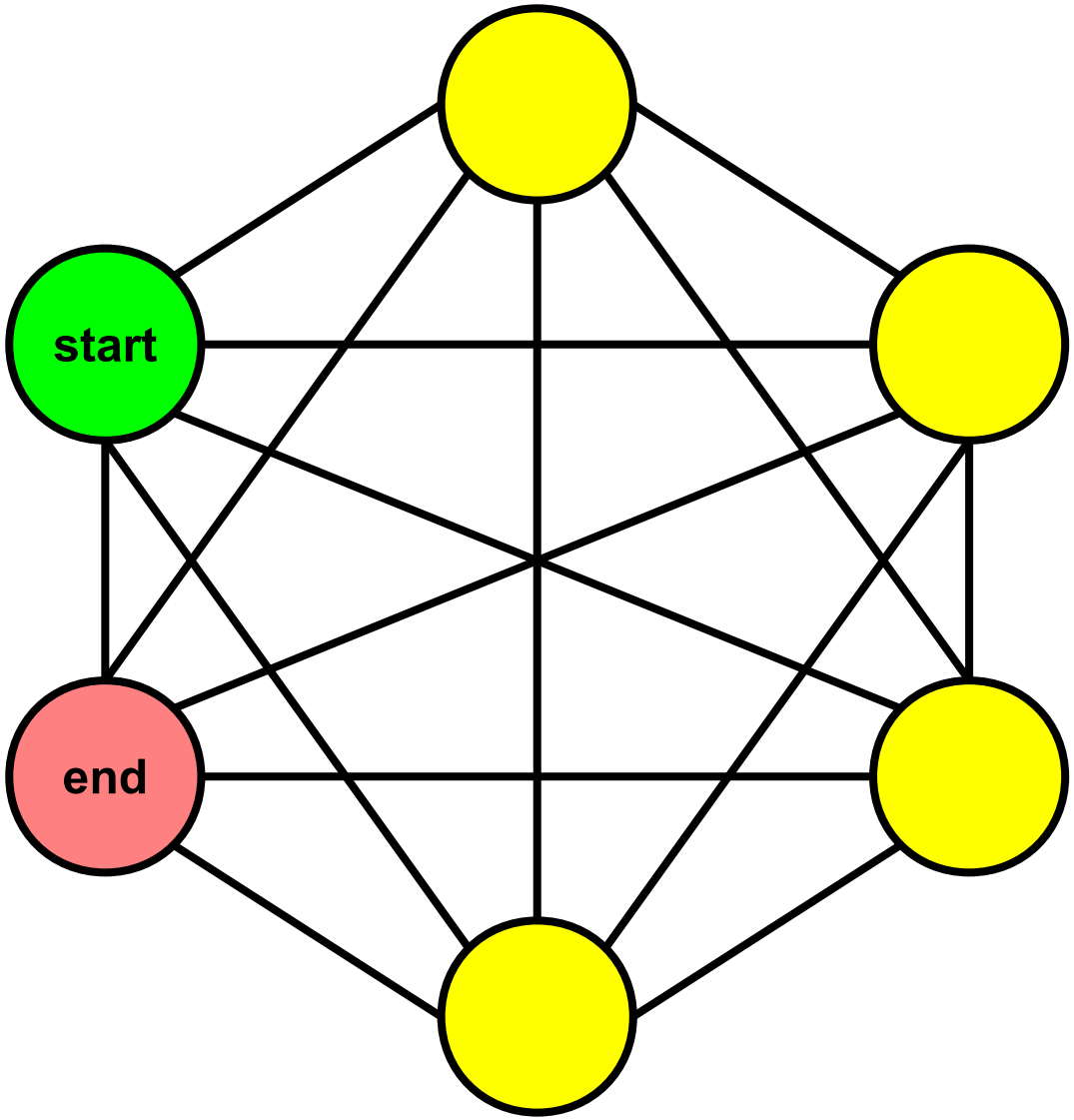
Examples of Problems in **P**

- 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


Regular Languages
CFLs
P
R
Undecidable Languages

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



How many 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

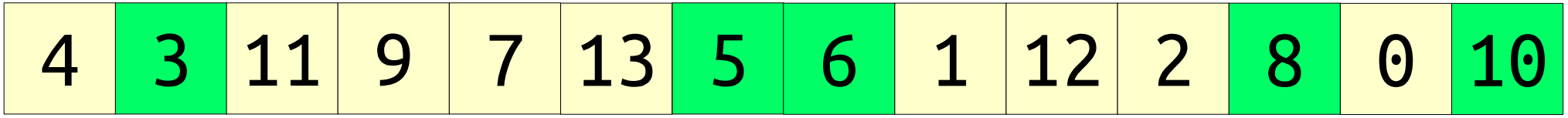
Does this Sudoku problem have a solution?

Verifiers – Again

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

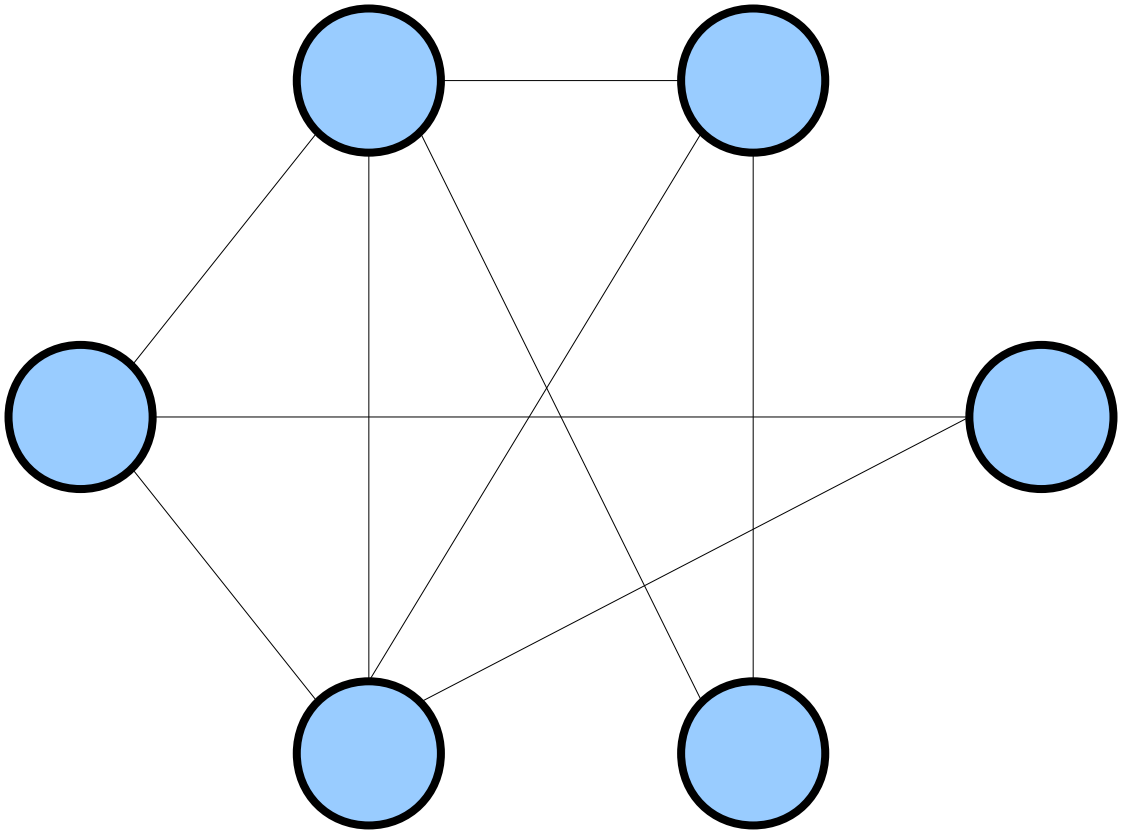
Is there an ascending subsequence of length at least 5?

Verifiers – Again



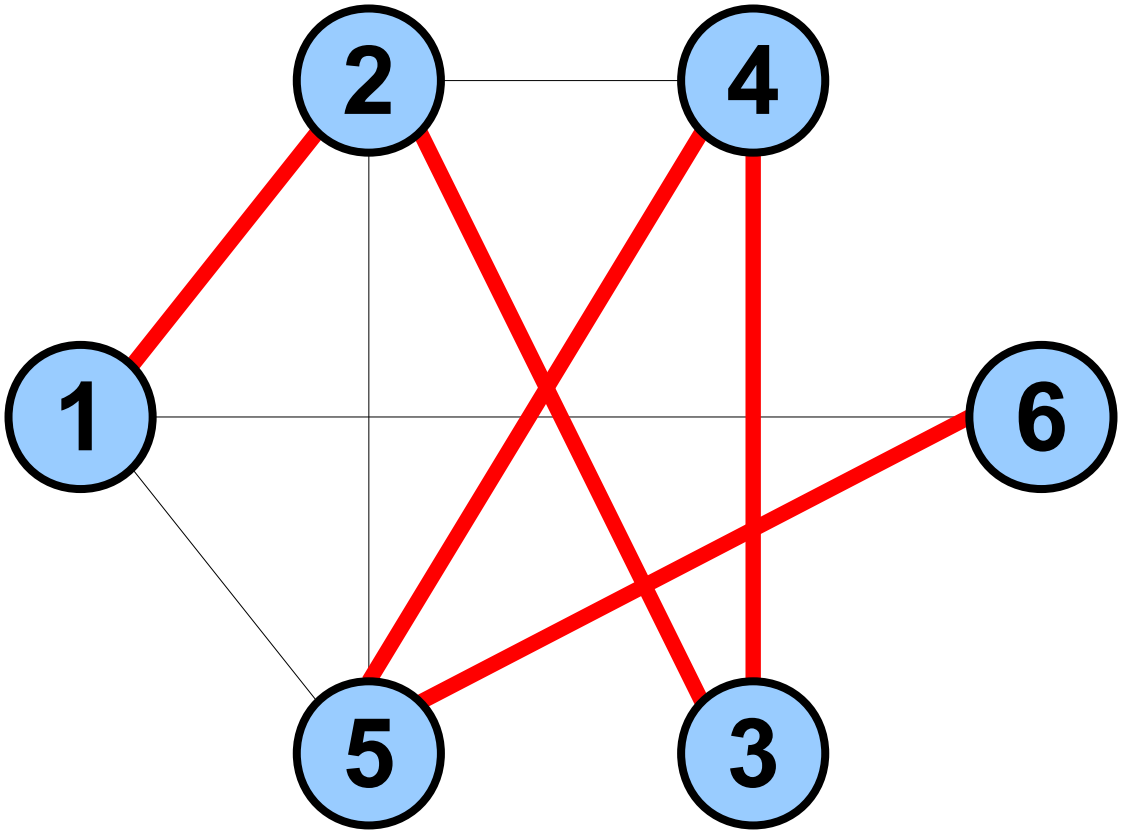
Is there an ascending subsequence of length at least 5?

Verifiers – Again



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

Verifiers – Again



Is there a 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.
  - $w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V$ 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.
  - $w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*$. V accepts $\langle w, c \rangle$.
  - V runs "efficiently" (its runtime is $O(|w|^k)$ for some $k \in \mathbb{N}$).
  - All strings in L have "short" certificates (their lengths are $O(|w|^r)$ for some $r \in \mathbb{N}$).

The Complexity Class **NP**

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

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

- 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:*** **NP** ϙ**R**.
  - ***Proof idea:*** If $L \in \textbf{NP}$, all strings in L have "short" certificates. Therefore, we can just try all possible "short" certificates and see if any of them work. (Showing **NP** is a strict subset of **R** requires some more advanced 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**?

And now…

The

Biggest Question

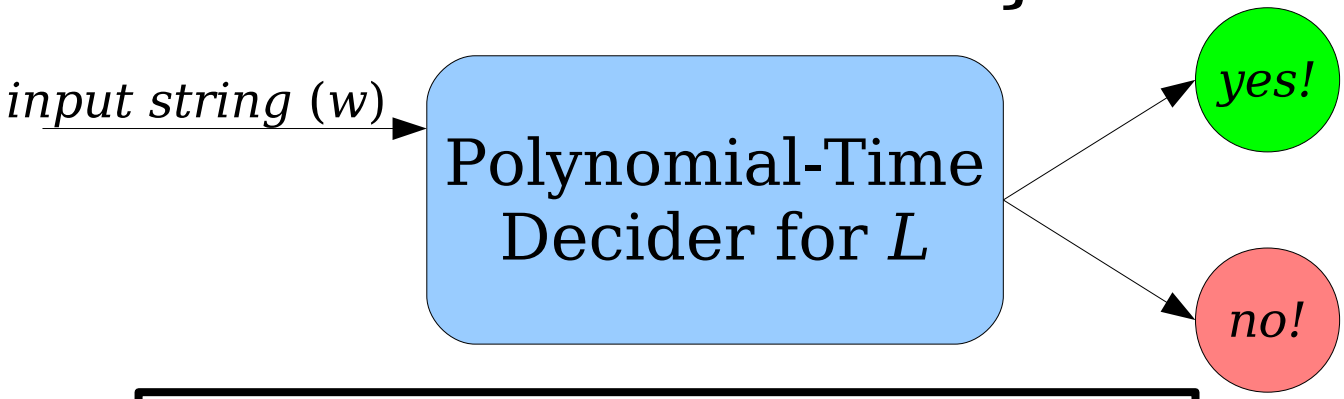
in

Theoretical Computer Science

P 'PNP

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

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

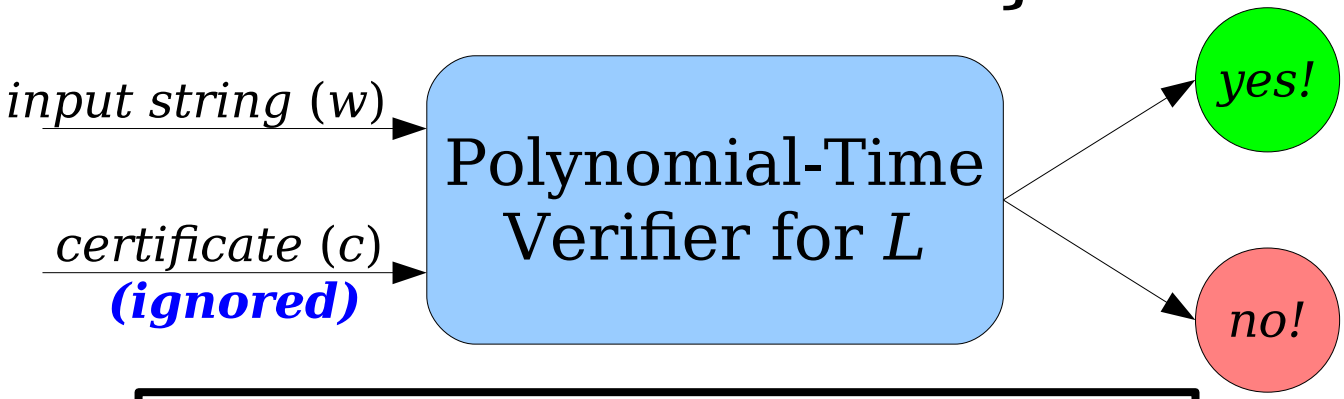



```
bool solveProblemL(string w) {

    do some work;
    return the answer;
}
```

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

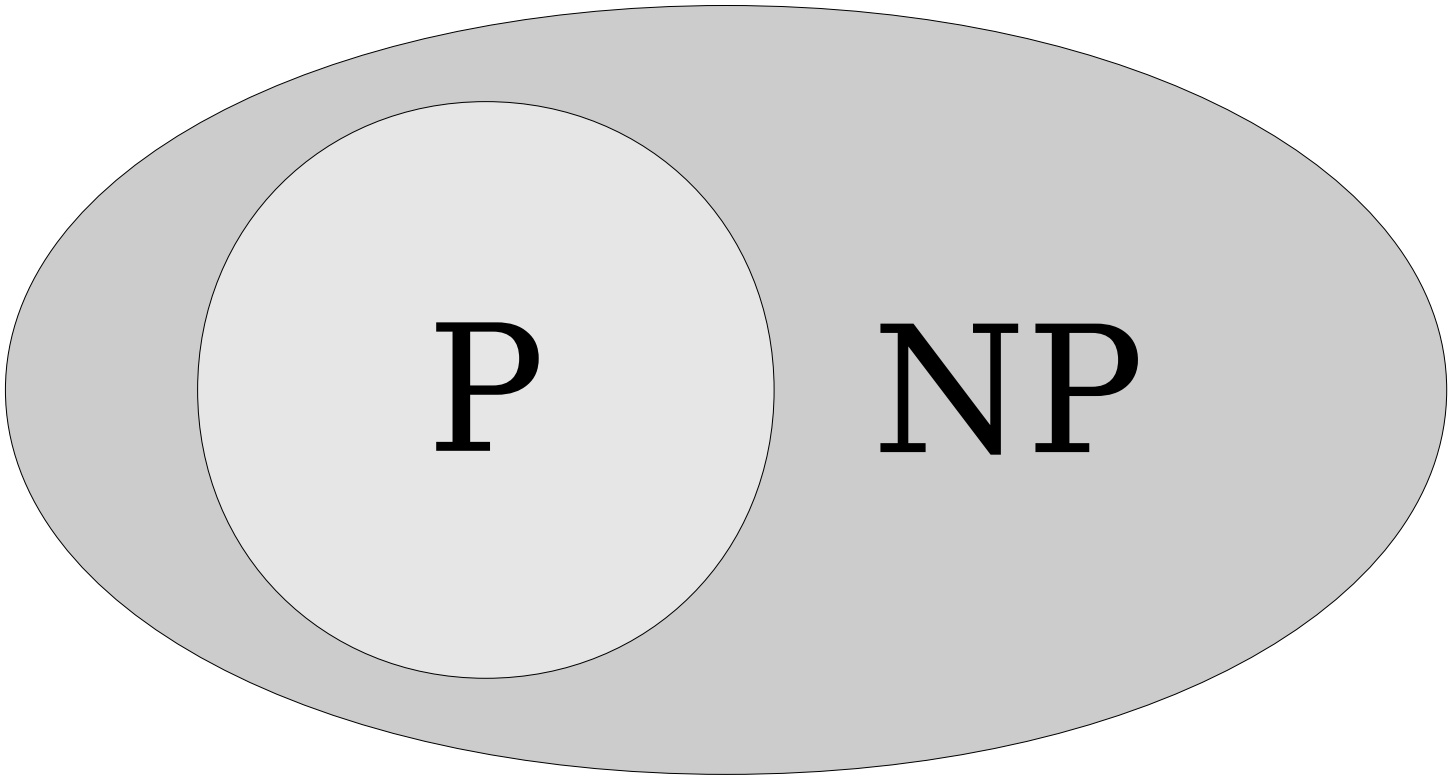
NP = { L | There is a polynomial-time verifier for L }



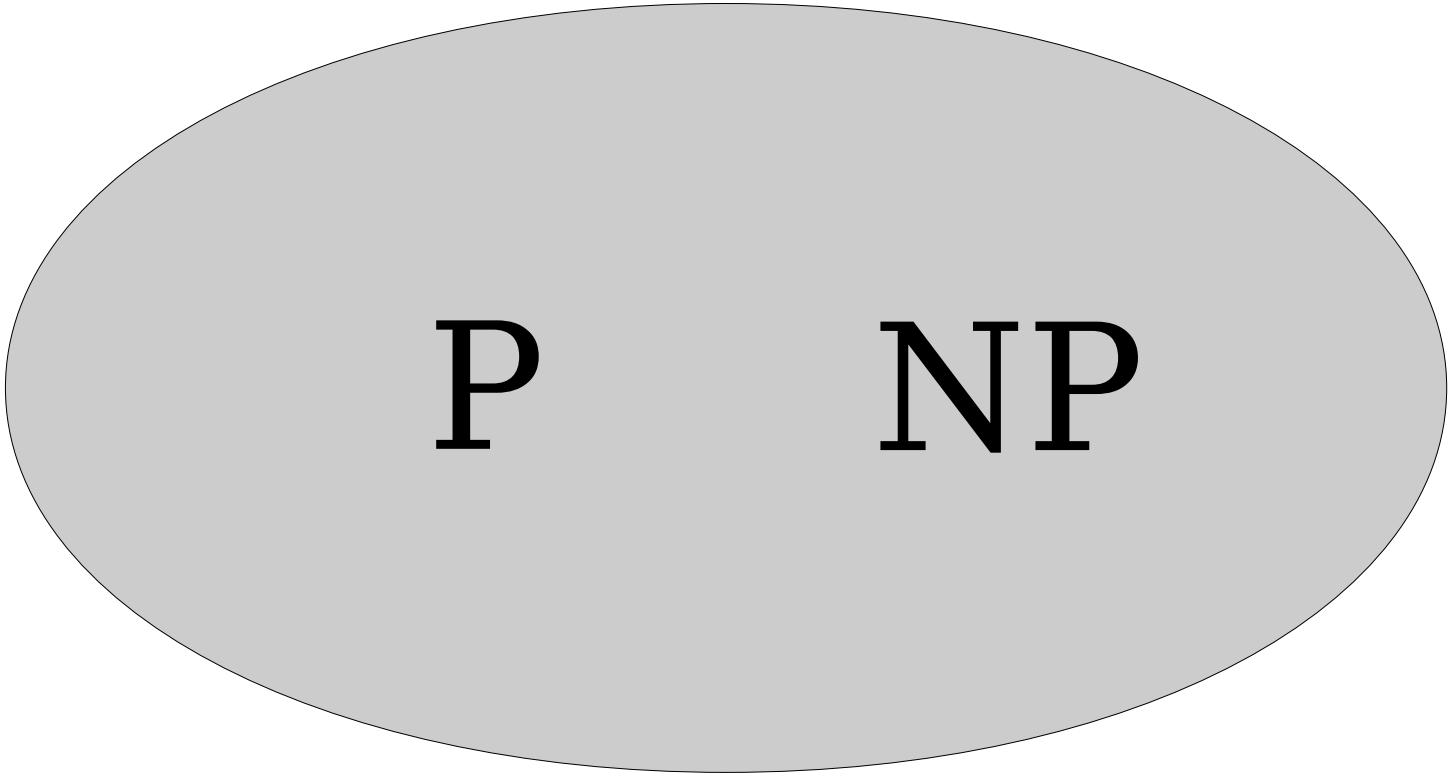

```
bool solveProblemL(string w, string c) {
    /* don't even look at c */
    do some work;
    return the answer;
}
```

P ⊆ NP

Which Picture is Correct?



Which Picture is Correct?



P ≟ **NP**

- The **P** ≟ **NP** question is the most important open 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 also 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** ≟ **NP** has proven ***extremely difficult***.
- In the past 50 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** ≟ **NP**.
  - A majority of computer scientists believe **P** ≠ **NP**, but it isn't an overwhelming majority.
- Interesting read: Interviews with leading thinkers about **P** ≟ **NP**:
  - https://www.cs.umd.edu/~gasarch/papers/poll.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** ̛P **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 ̛PNP *problem.)

What do we know about **P** ≟ **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 simulate other TMs.
  - ***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!