Unsolvable Problems

Part Three

Recap from Last Time

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
  - If V does not accept $\langle w, c \rangle$, then either
    - $w \in L$, but you gave the wrong c, or
    - $w \notin L$, so no possible c will work.

Verifiers and **RE**

- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof idea:*** Build a recognizer that tries every possible certificate to see if $w \in L$.
- ***Proof sketch:*** Show that this TM is a recognizer for L:

M = "On input w:
    For $i = 0$ to ∞
        For each string c of length i:
            Run V on $\langle w, c \rangle$.
            If V accepts $\langle w, c \rangle$, M accepts w."

Verifiers and **RE**

- ***Theorem:*** If $L \in$ **RE**, then there is a verifier for L.
- ***Proof sketch:*** Let L be an **RE** language and let M be a recognizer for it. Then show that this is a verifier for L:

V = "On input $\langle w, n \rangle$, where $n \in \mathbb{N}$:
Run M on w for n steps.
If M accepts w within n steps, accept.
If M did not accept w in n steps, reject."

Nondeterministic TMs

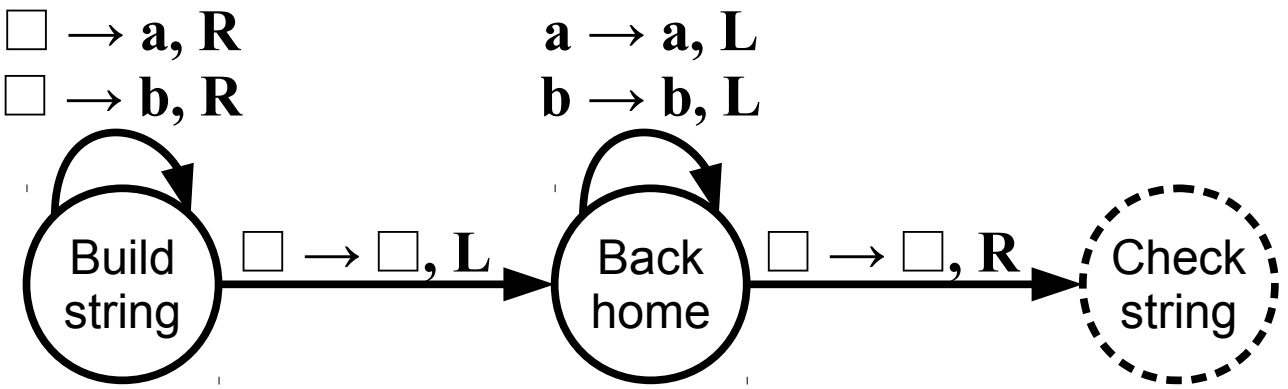
- A ***nondeterministic Turing machine*** (or ***NTM***) is a Turing machine in which there can be zero or multiple transitions defined at each state.
- Nondeterministic TMs do not have ε-transitions; they have to read or write something and move the tape at each step.
- As with NFAs, NTMs accept if any path accepts. In other words, an NTM for a language L is one where

  $w \in L$ iff there is some series of choices N can make that causes N to accept w.

- In particular, if $w \in L$, N only needs to accept w along one branch. The rest can loop infinitely or reject.

Guessing an Arbitrary String

- Here's how an NTM can guess an arbitrary string, then go do something with it:



- As a high-level description:

> N = "On input w:
> Nondeterministically guess a string $x \in \Sigma^*$.
> Deterministically check whether [...]"

NTMs and DTMs

- ***Theorem:*** If $L \in \mathbf{RE}$, then there is an NTM for L.
- ***Proof Sketch:*** Every deterministic TM (DTM) can be thought of as an NTM with no nondeterminism, so if L is the language of a DTM, it's also the language of an NTM. ■

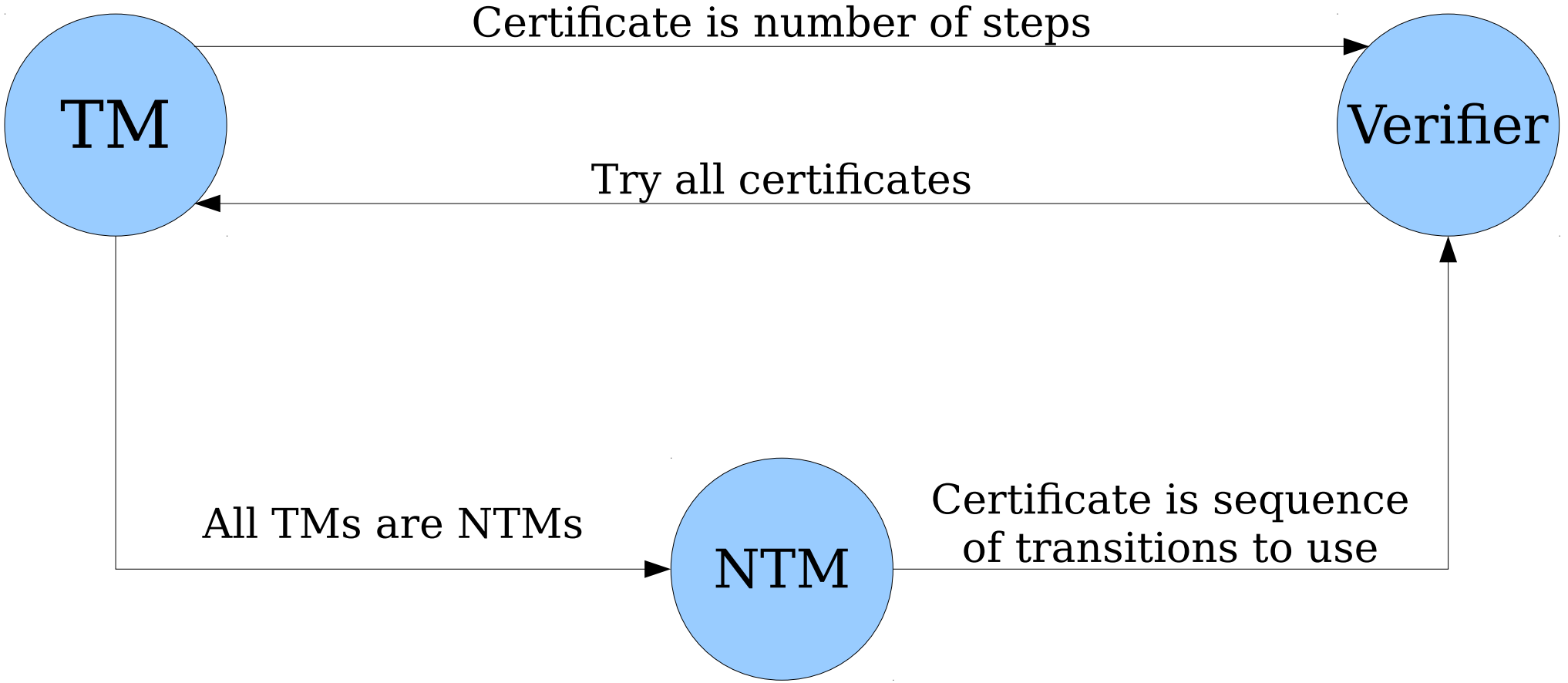
NTMs and DTMs

- ***Theorem:*** If L is the language of an NTM, then $L \in \mathbf{RE}$.
- ***Proof Sketch:*** Let N be an NTM for L. Then we can prove that this is a verifier for L:

V = "On input $\langle w, T \rangle$, where T is a sequence of transitions:

- Run N on w, following transitions in the order specified in T.
- If any of the transitions in T are invalid or can't be followed, reject.
- If after following the transitions N accepts w, accept; otherwise reject.

Three Views of **RE**



Why This Matters

- We now have three perspectives on the **RE** languages:
  - They're languages where a TM can *search* for proof that a string in the language.
  - They're languages where a verifier can *check* a proof that a string is in the language.
  - They're languages where an NTM can *guess* a proof that a string is in the language.
- All of this comes back to the notion of *proving strings in the language*: you might not be able to *determine* whether a string is in an **RE** language, but if a string is in an **RE** language, there is some way to prove it.

New Stuff!

Finding Non-**RE** Languages

Self-Reference and **RE**

- Self-reference is the main technique we'll use to find non-**RE** languages.
- However, things are a bit more complicated when finding non-**RE** languages than finding non-**R** languages.
- Why?
  - When talking about deciders, we assume we have a magic subroutine that always produces the answer we want.
  - **RE** is the class of problems with recognizers, NTMs, and verifiers. These are much harder to reason about.

The Unhalting Problem

- Consider this problem:

  Given a TM *M* and a string *w*, does *M* loop on *w*?

- As a formal language:

  ***LOOP* = { ⟨*M*, *w*⟩ | *M* is a TM that loops on *w* }**

- How hard of a problem is this to solve?

The Unhalting Problem

- Intuitively, would we expect this language to belong to class **R**?
  - ***No:*** The only general way to figure out what a TM will do is to run it.
  - Can formalize this by building a program that asks if it will loop and does the opposite.
- Intuitively, would we expect this language to belong to class **RE**?
  - ***No:*** If you were *convinced* that a TM will go into an infinite loop, what could you do to prove this to me?

Self-Reference and Verifiers

- Assume for the sake of contradiction that $LOOP \in \mathbf{RE}$.
- This means that there's a verifier for *LOOP*.
- In software, that would be a function like this one:

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate)
```

This function is our verifier:

- If `program` loops on `input`, then there is some choice of `certificate` that causes this function to return true.
- If `program` halts on `input`, this function always returns false, regardless of what `certificate` is.
- The function always returns a value.

```
bool imConvincedWillLoop(string program,
                         string input,
                         string certificate) {
    /* … implementation … */
}

int main() {
  string me = mySource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedWillLoop(me, input, c)) {
        accept();
      }
    }
  }
}
```

What happens if...

... this program loops on its input?

There is a certificate that proves it loops.

So the program halts after it finds that certificate!

... this program halts on its input?

There is no certificate that proves it loops.

So the program loops infinitely!

Self-Reference and **RE**

- The proof template for showing undecidability via self-reference is the following:
  - Build a machine that asks what it is about to do.
  - Based on the answer, have the machine do the exact opposite.
- The proof template for showing *unrecognizability* via self-reference is the following:
  - Build a machine that tries to *prove* it will do something.
  - If it *proves* that it will, have it do the opposite.
  - If it can't *prove* that it will, it will loop infinitely. Design the machine so that looping infinitely causes it to behave incorrectly.

Theorem: $LOOP \notin \mathbf{RE}$.

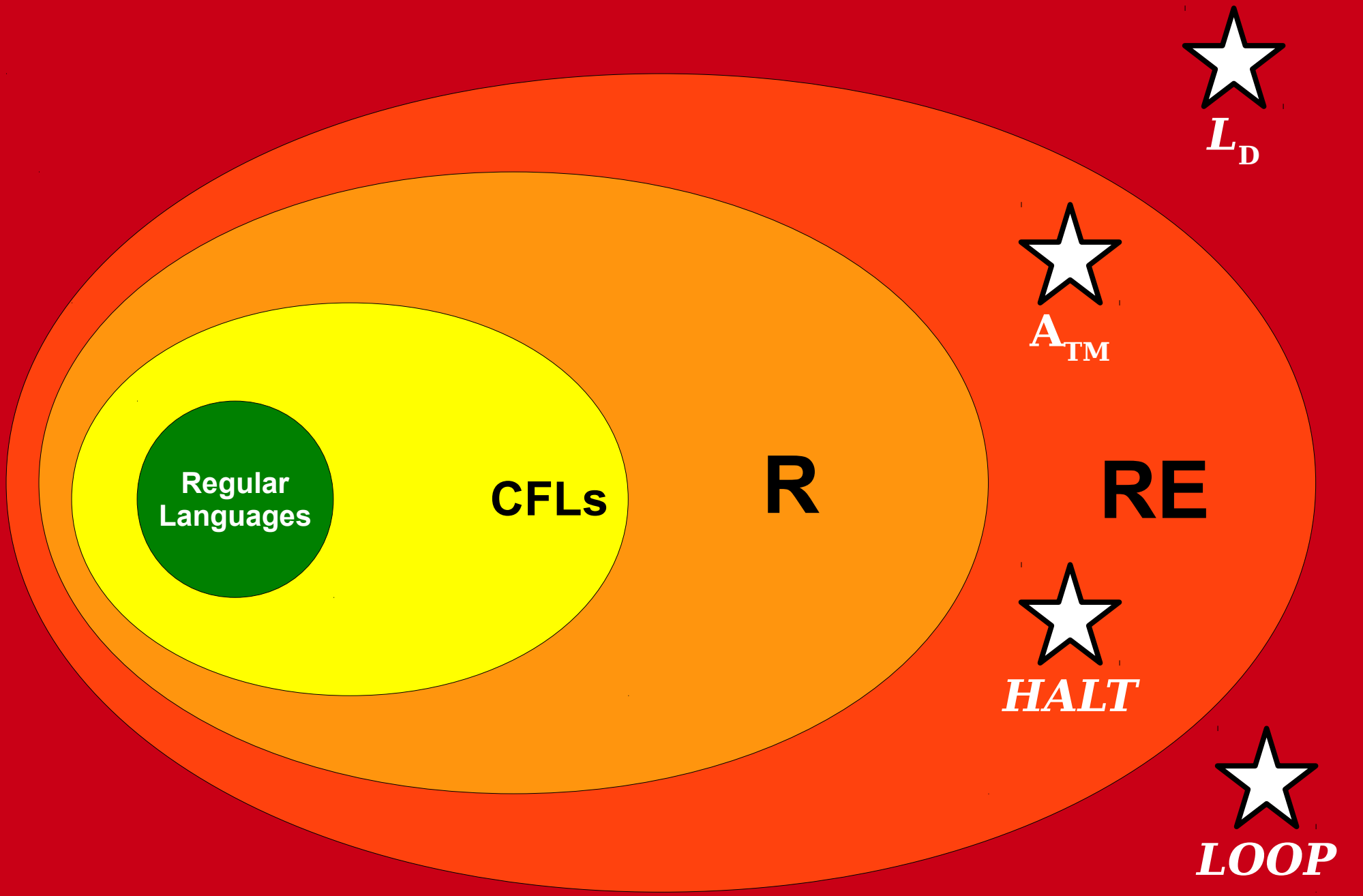
Proof: By contradiction; assume that $LOOP \in \mathbf{RE}$. Then there is some verifier V for $LOOP$. This verifier has the property that if M is a TM that loops on some string w, there is a certificate c such that V accepts $\langle M, w, c \rangle$, and if M halts on w, V will never accept $\langle M, w, c \rangle$ for any certificate c.

Given this, we could then construct the following TM:

M = "On input w:
    Have M obtain its own description, $\langle M \rangle$.
    For all strings c:
        If V accepts $\langle M, w, c \rangle$, accept.

Choose any string w and trace through the execution of the machine. If V ever accepts $\langle M, w, c \rangle$, we are guaranteed that M loops on w, but in this case we find that M accepts w, a contradiction. If V never accepts $\langle M, w, c \rangle$, then we are guaranteed that M halts on w, but in this case we find that M loops infinitely on w, a contradiction.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $LOOP \notin \mathbf{RE}$. ■


L_D
A_{TM}
Regular Languages
CFLs
R
RE
HALT
LOOP
All Languages

$LOOP \notin \mathbf{RE}$

- The fact that $LOOP \notin \mathbf{RE}$ gives us a powerful intuition:

  There is no general way to prove that a TM will loop on a particular input.

- This is a really useful intuition going forward – it will help you get a better sense for when a problem is likely to be unrecognizable.

Non-**RE** Languages

- If a problem is not in **RE**, then there must be instances of the problem where the answer is “yes,” but there is no way to prove that the answer is “yes.”
- Not only will rational thought not always *discover* the truth, rational thought is not powerful enough to even *confirm* the truth.

There are true statements that are not provable!

Where We Stand

- We've just done a crazy, whirlwind tour of computability theory:
  - ***The Church-Turing thesis*** tells us that TMs give us a mechanism for studying computation in the abstract.
  - ***Universal computers*** – computers as we know them – are not just a stroke of luck. The existence of the universal TM ensures that such computers must exist.
  - ***Self-reference*** is an inherent consequence of computational power.
  - ***Undecidable problems*** exist partially as a consequence of the above and indicate that there are statements whose truth can't be determined by computational processes.
  - ***Unrecognizable problems*** also exist partially through self-reference and indicate that there are limits to mathematical proof.

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.

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”

A Decidable Problem

- ***Presburger arithmetic*** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \rightarrow x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \wedge \forall y.\ (P(y) \rightarrow P(y + 1))) \rightarrow \forall x.\ P(x)$
- Given a statement, it is decidable whether that statement can be proven from the laws of Presburger arithmetic.
- Any Turing machine that decides whether a statement in Presburger arithmetic is true or false has to move the tape head at least $2^{2^{cn}}$ times on some inputs of length n (for some fixed constant c).

For Reference

- Assume $c = 1$.

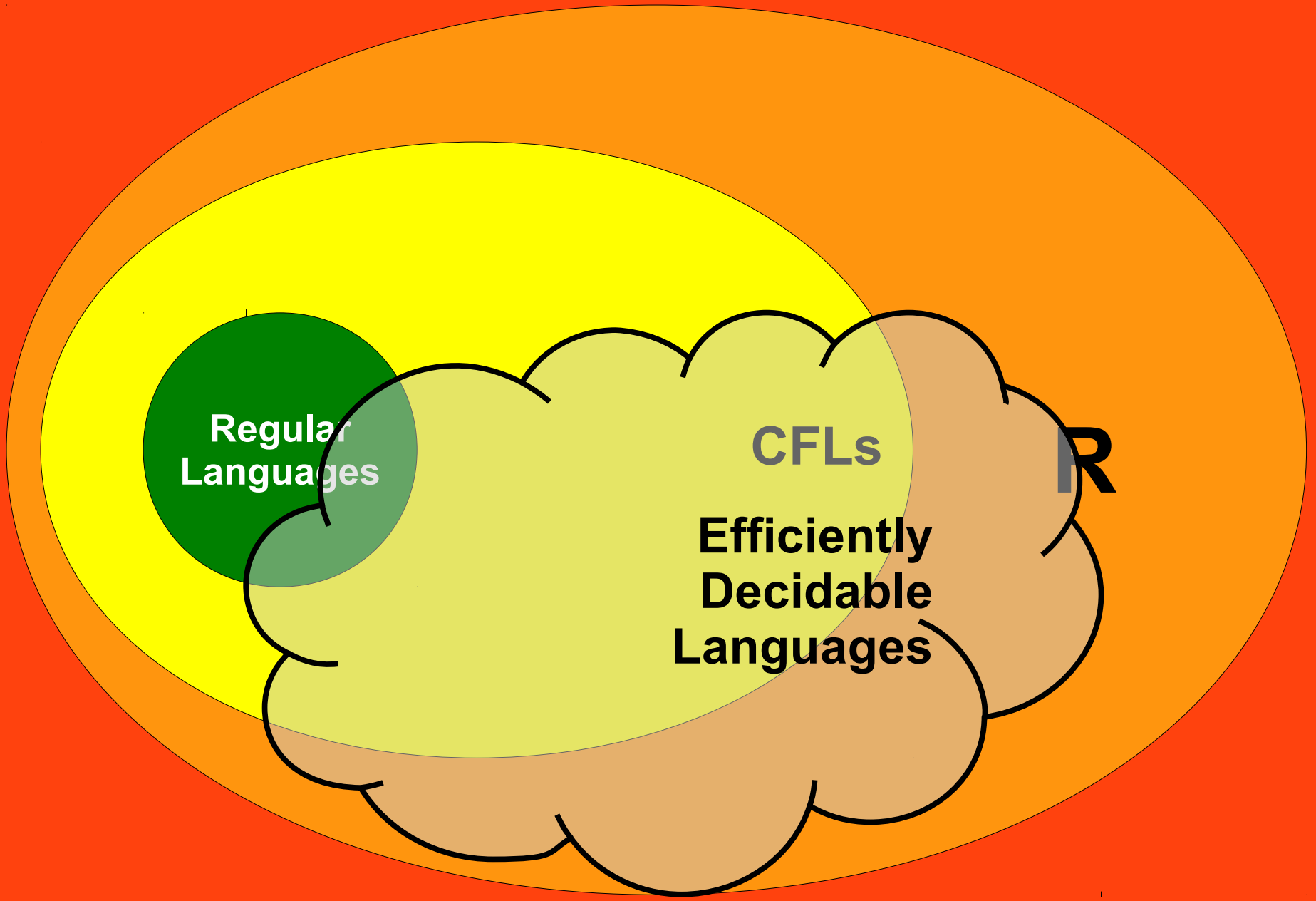
$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$
$$2^{2^4} = 65536$$
$$2^{2^5} = 18446744073709551616$$
$$2^{2^6} = 340282366920938463463374607431768211456$$

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.


Regular Languages
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

Time-Out for Announcements!

Midterm Grading

- You're done with midterms for CS103! Hooray!
- The TAs will be grading the second midterm over the long weekend. We'll have it graded and will release solutions and stats on Wednesday.

Your Questions

“It seems like every quarter has this horrible overwhelming part in the middle. People tell me to just suffer though it because it'll be over in a few weeks. We're supposed to be here for several years, though. How can I make it sustainable?”

You can end up feeling overwhelmed if

- you have too many things to do,
- you have to do all of them well,
- you don't enjoy them,
- you have limited time to do them, and
- this process repeats.

Try addressing each of these independently. If you can address the root causes of each, you will probably end up a lot happier.

“I read an article about artificial intelligence that said that within the century we will probably develop AI that is smarter than us, and by self-improvement we will end up with superintelligent machines that can change everything. Is this possible?”

I don't see any reason why this couldn't happen. I doubt this will happen any time soon, though.

“How is the final grading going to happen in this class? The raw percentages on the assignments and the corresponding percentiles are very different.”

I use raw scores for everything – I only curve raw total scores. Only then do I look at percentiles.

"CANDIDATE FOR CUTEST ANIMAL"

slow Loris are awesome. ☺

Back to CS103!

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

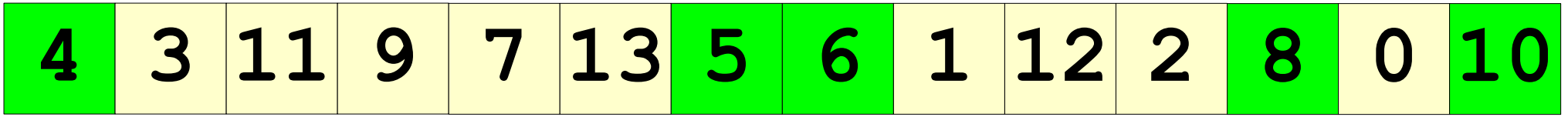
- Suppose that we have a decider D for some language L.
- How might we measure the complexity of D?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Amount of time required.
  - Number of times a given state is entered.
  - Number of times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

What is an efficient algorithm?

Searching Finite Spaces

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

A Sample Problem



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

A Sample Problem

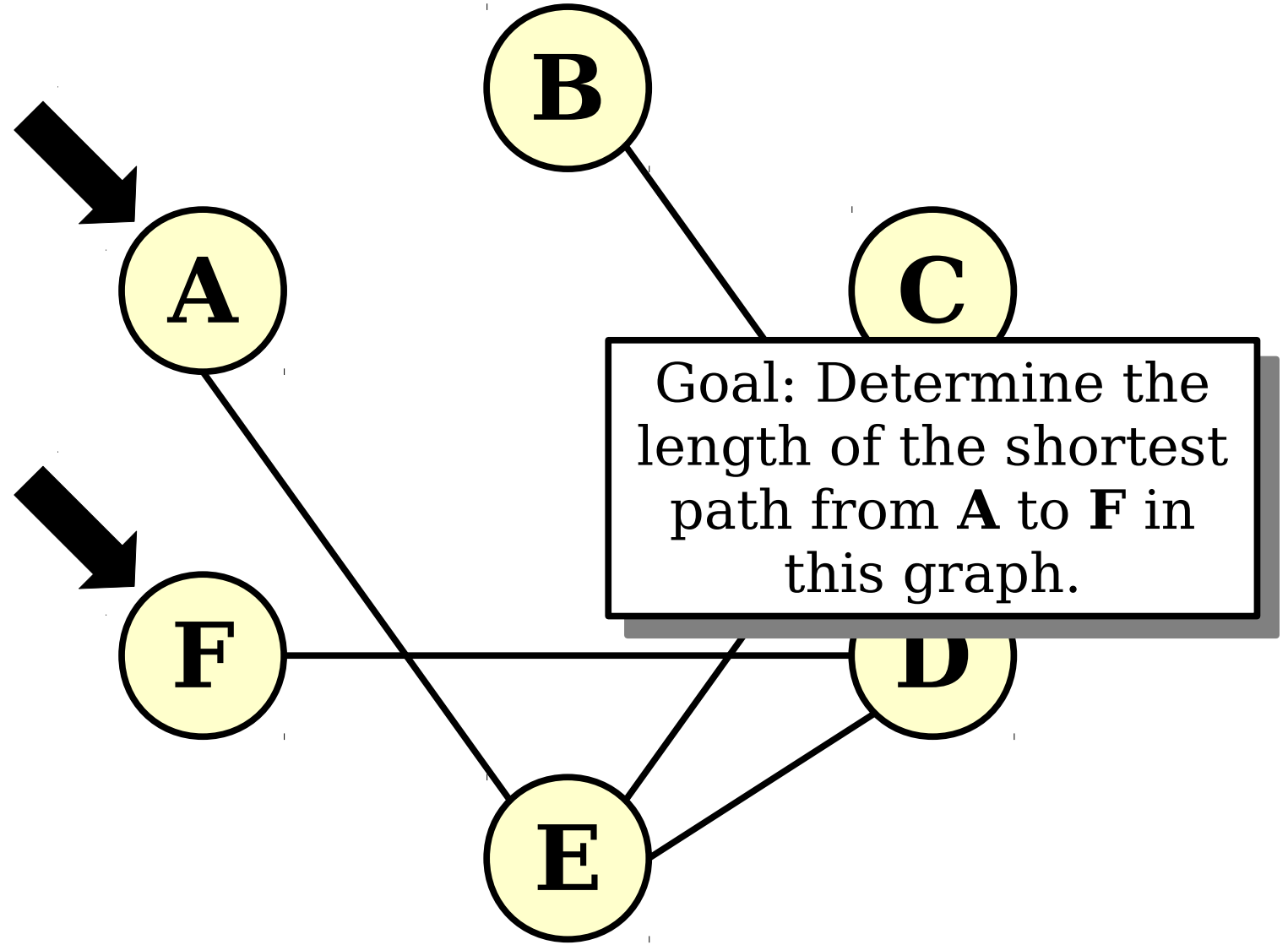
4	3	11	9	7	13	5	6	1	12	2	8	0	10

1	1	2	2	2	3	2	3	1	4	2	4	1	5

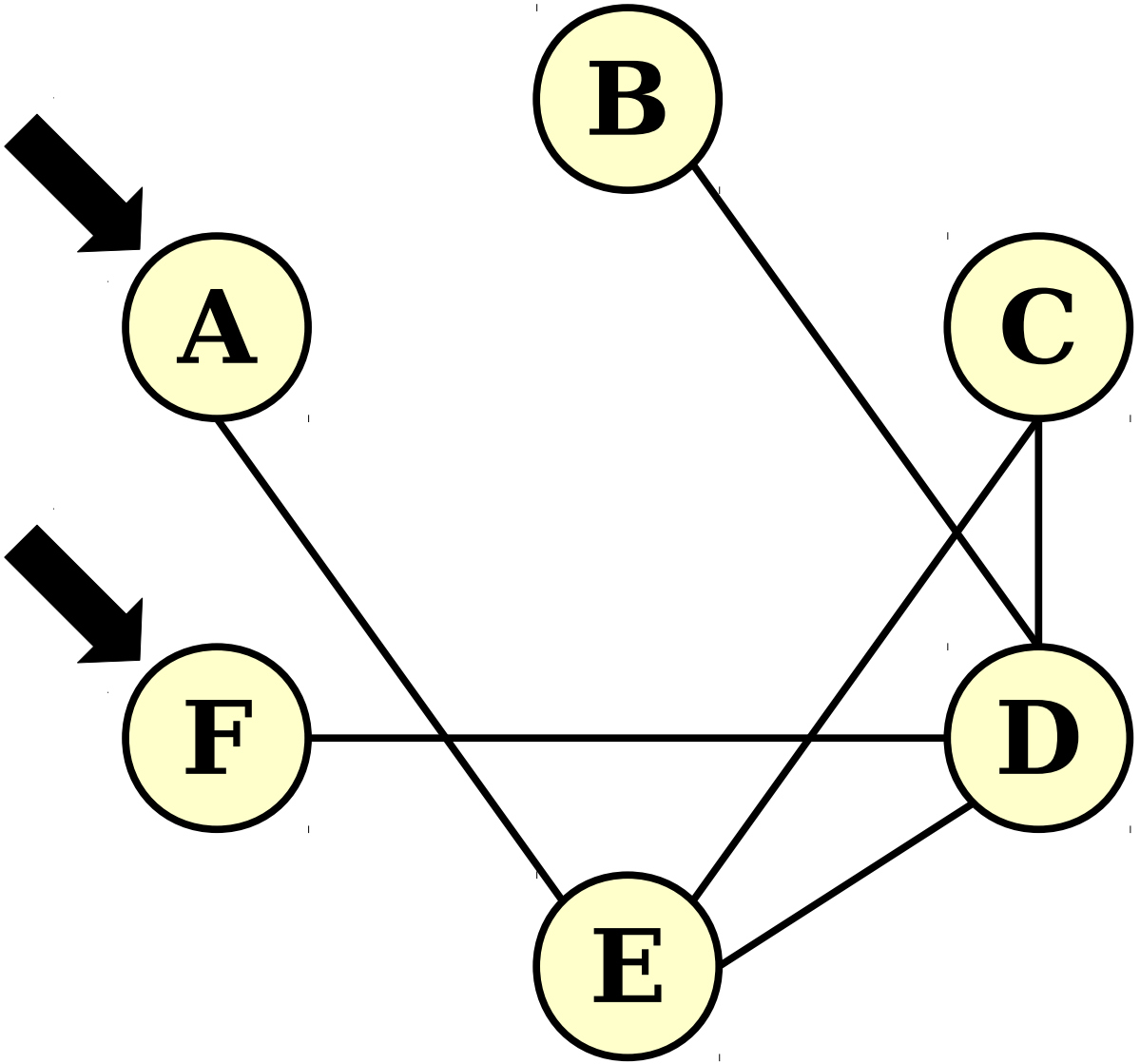
How many elements of the sequence do we have to look at when considering the kth element of the sequence? $\boldsymbol{k - 1}$

Total runtime is
$1 + 2 + \ldots + (n - 1) = \mathbf{O(n^2)}$

Another Problem



Another Problem

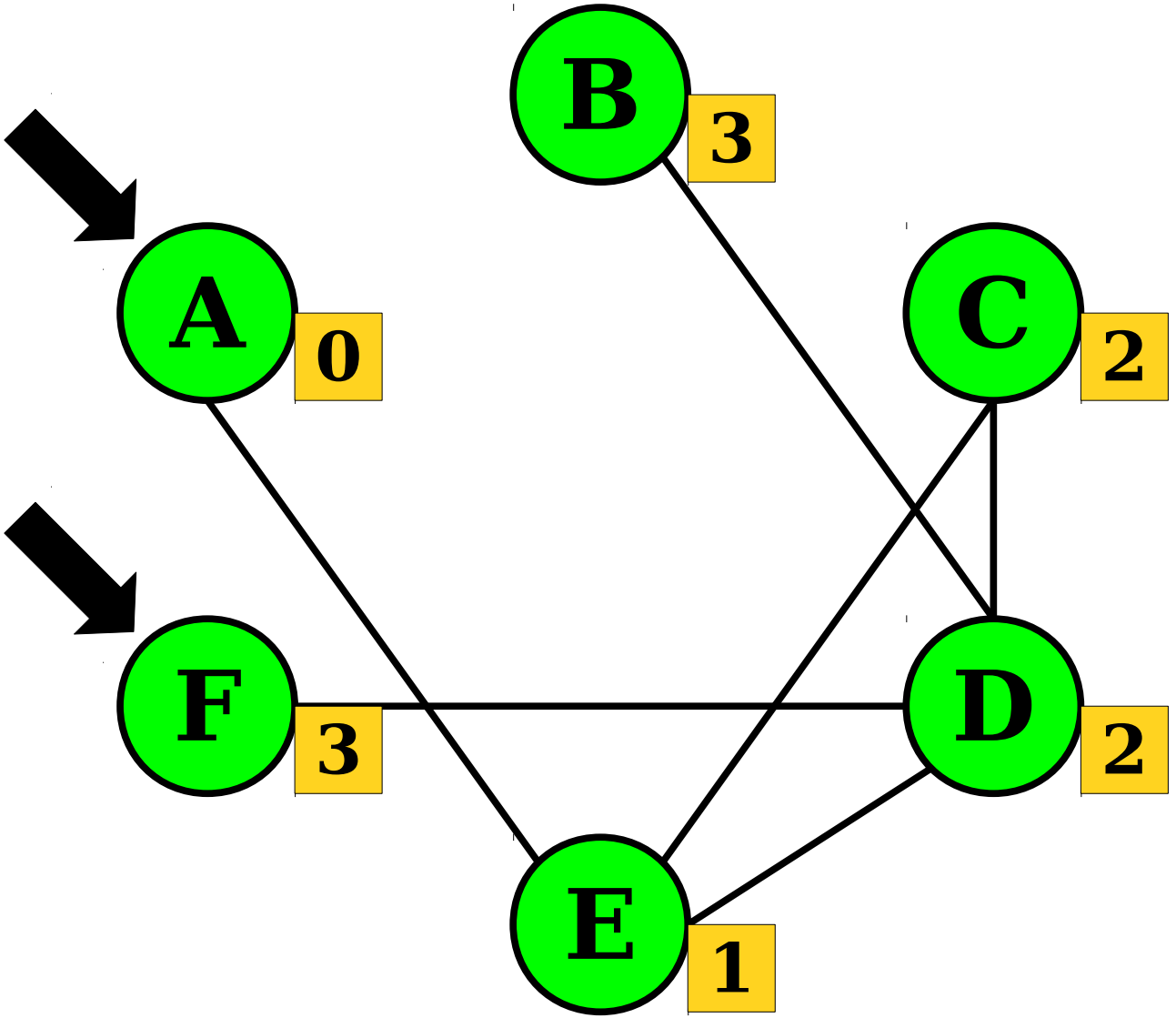


Number of possible ways to order a subset of n nodes is **$O(n \cdot n!)$**

Time to check a path is **$O(n)$**.

Runtime: **$O(n^2 \cdot n!)$**

Another Problem



With a precise analysis, runtime is **O(*n* + *m*)**, where *n* is the number of nodes and *m* is the number of edges.

For Comparison

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

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 iff it can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Like the Church-Turing thesis, this is ***not*** a theorem!

It's an assumption about the nature of efficient computation, and it is somewhat controversial.

The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 - 2n^2 + 4n$
  - $n \log \log n$
- "Efficient" runtimes:
  - $n^{1,000,000,000,000}$
  - 10^{500}
- Inefficient runtimes:
  - 2^n
  - $n!$
  - n^n
- "Inefficient" runtimes:
  - $n^{0.0001 \log n}$
  - 1.000000001^n

Why Polynomials?

- Polynomial time *somewhat* captures efficient computation, but has a few edge cases.
- However, polynomials have very nice mathematical properties:
  - The sum of two polynomials is a polynomial. (Running one efficient algorithm after the other gives an efficient algorithm.)
  - The product of two polynomials is a polynomial. (Running one efficient algorithm a “reasonable” number of times gives an efficient algorithm.)
  - The *composition* of two polynomials is a polynomial. (Using the output of one efficient algorithm as the input to another efficient algorithm gives an efficient algorithm.)

The Complexity Class **P**

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

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

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