CSE 417 Autumn 2025

Lecture 27: Proving things hard

Glenn Sun

Logistics

Practice final has been released!

This Friday's lecture: Review session

The last Concept Check quiz is to **complete the course evaluation**.

- Is anonymous (we see who completed it, but not who said what)
- Due Monday night (after finals) 11:59pm
- Not open yet (?)

What is an algorithm?

What is an algorithm?

First day of class:

- **Algorithm:** A list of unambiguous instructions to solve a class of computational problems.

What does it mean for a problem to be computational?

What are instructions?

If our goal is to prove something about “all algorithms”, we need to know precisely what an algorithm formally is.

Rigorously defining “algorithm”

Here are a modern-day attempt:

🤔

- An **algorithm** is a Java program that successfully compiles (possibly non-terminating or having runtime errors)

✅
- A **problem** is a mathematical function that maps binary strings to binary strings. (Inputs/outputs encoded in binary.)

🤔
- The algorithm **solves the problem** if for all inputs, when running the algorithm, it terminates, has no runtime errors, and matches the expected output of the problem.

Rigorously defining “algorithm”

Our first definition required:

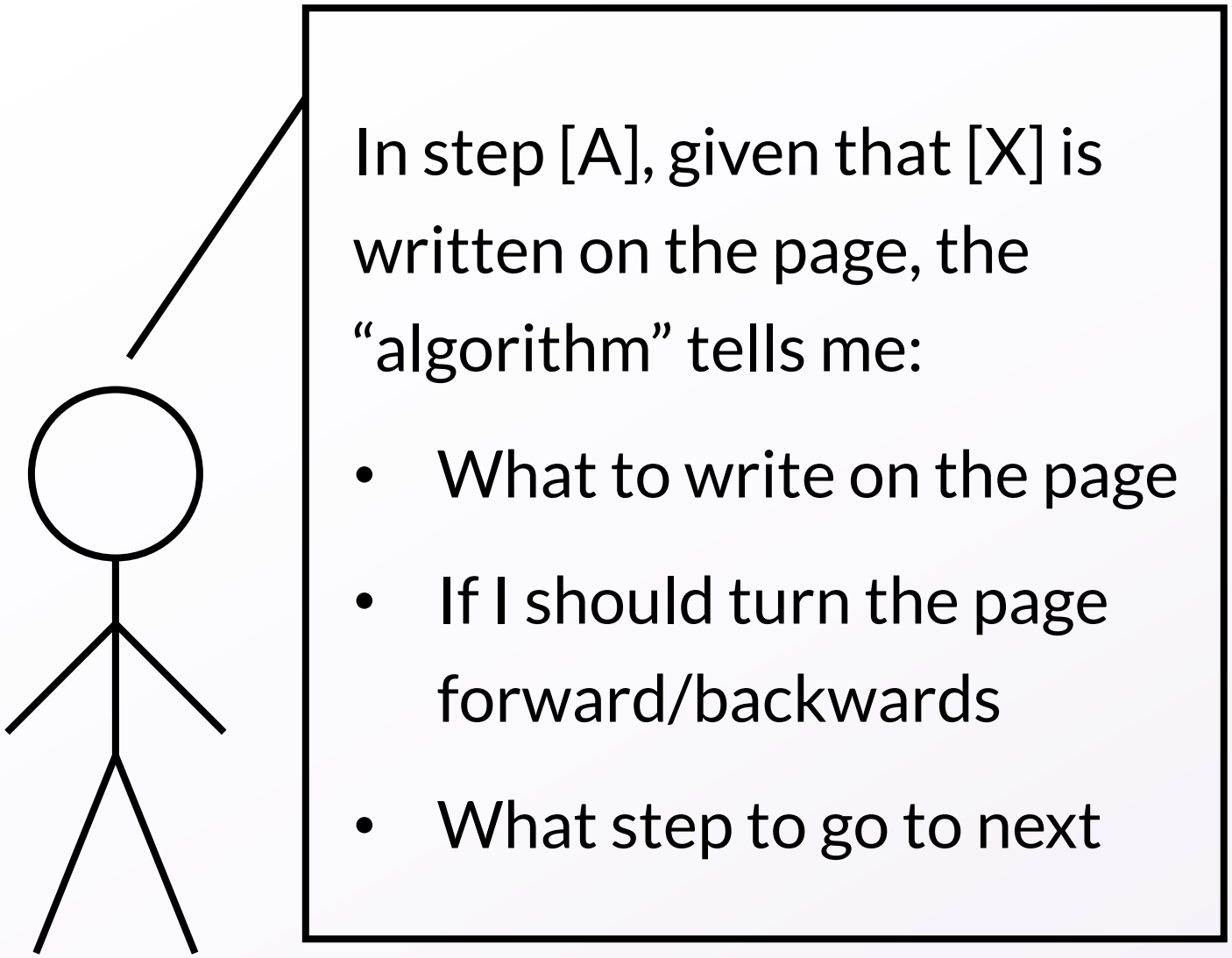
- Understanding the Java spec
- Understanding how a computer runs Java code

This is mathematically rigorous.

But this is extraordinarily complicated for defining “algorithm”!

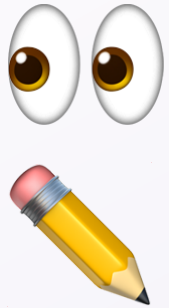
1936: Alan Turing invents the **Turing machine**!

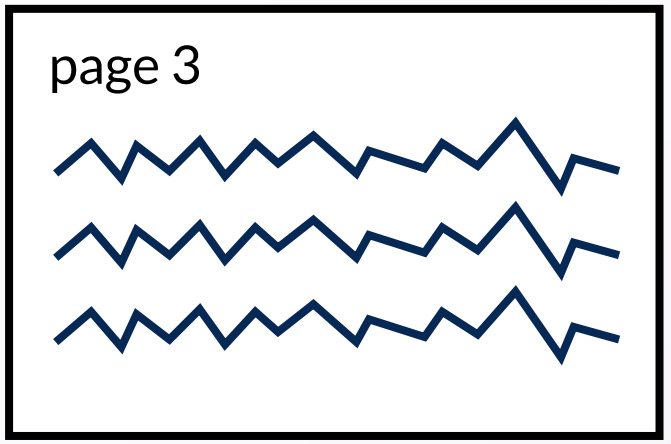
Turing machines

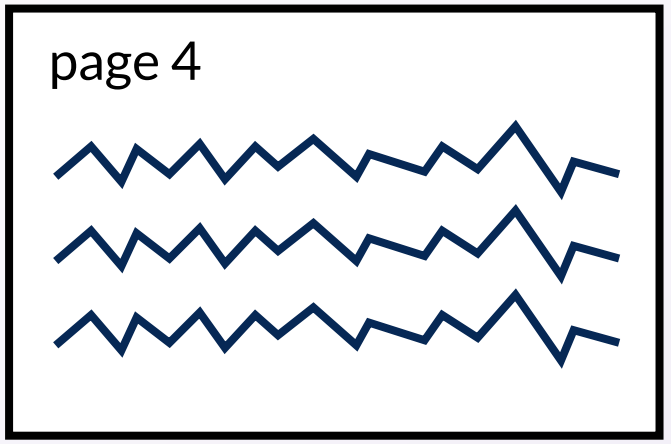


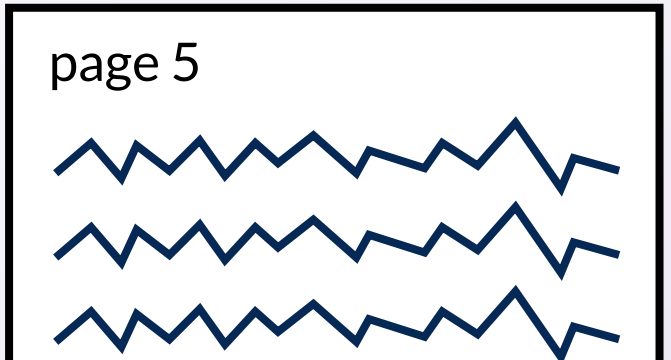
In step [A], given that [X] is written on the page, the "algorithm" tells me:

- What to write on the page
- If I should turn the page forward/backwards
- What step to go to next

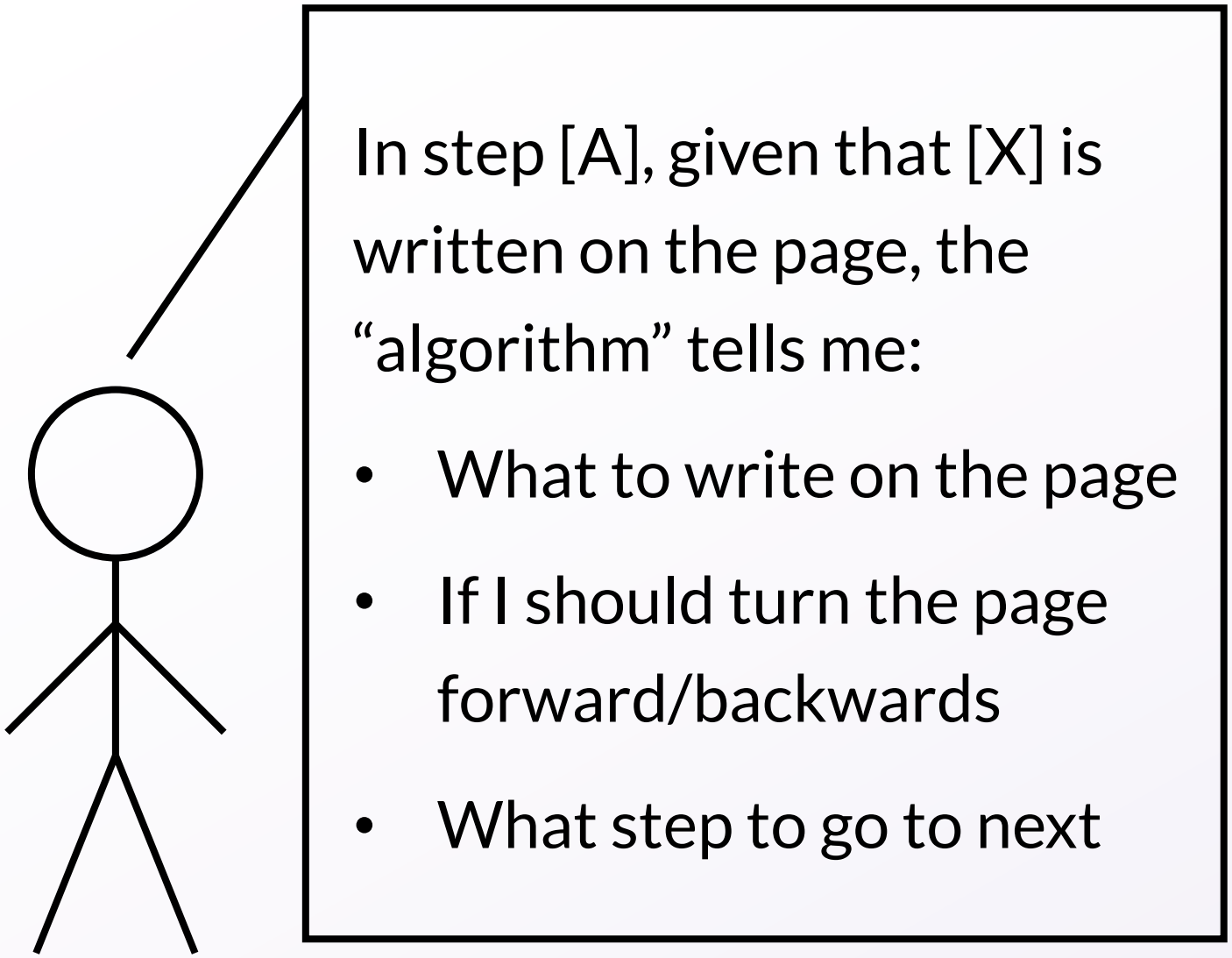






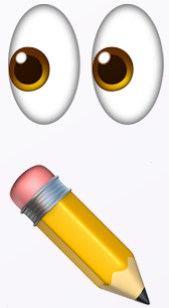


Turing machines



In step [A], given that [X] is written on the page, the "algorithm" tells me:

- What to write on the page
- If I should turn the page forward/backwards
- What step to go to next



page 32495

1

page 32496

0

page 32497

0

Mathematical formalism of Turing machines

Let $Q \subseteq \mathbb{N}$ be a finite set of states ("steps" of the algorithm).

A Turing machine consists of:

- An infinitely long **tape** (paper): an infinite list of elements from $\{0, 1, \_\}$, with only finitely many 0's and 1's.
- The **read/write head** (eyes): an index on the tape
- The **current state** (the step you're on): an element of Q

Mathematical formalism of Turing machines

An **algorithm** or **program** on a Turing machine is a function (often called a transition function):

$$\delta : Q \times \{0, 1, \_\} \rightarrow \{0, 1, \_\} \times \{L, R, N\} \times Q$$

Inputs:

- current state
- what's written on the tape at the read/write head

Outputs:

- what to write
- how to move the read/ write head
- new state

Mathematical formalism of Turing machines

Running a Turing machine on an input means:

- The tape starts with the input written on it
- The read/write head starts at index 0
- The state starts at a special state q_{start}.
- We repeatedly apply the transition function δ until the state is at a special state q_{end} (or possibly never terminate).
- Afterwards, the contents of the tape form the output.

Mathematical formalism of Turing machines

Recall that a **problem** is a function $f : \{0,1\}^* \to \{0,1\}^*$.

set of all binary strings

A Turing machine **solves** a problem f if for all inputs $x \in \{0,1\}^*$, upon running the Turing machine on input x, the Turing machine terminates and outputs $f(x)$.

Equivalence of computational models

There exists a Java program solving a problem f

if and only if

There exists a Turing machine solving f

This property is known as being "**Turing-complete**".

Why Turing machines?

A Turing machine is a simple model of a computer.

- Hard to write algorithms for, because there are so few features.
- Easy (-er) to use whenever talking about “all algorithms”, because there are so few features!

NP-hardness

SAT is NP-hard

Recall that a decision problem A is **NP-hard** if *any* problem in NP can be encoded as an instance of A.

Before we introduced Turing machines, it was very hard to talk about *all* problems in NP.

Now, we can prove:

Cook–Levin Theorem: SAT is NP-hard.

Non-deterministic Turing machines

In our transition function

$$\delta : Q \times \{0, 1, \_\} \to \{0, 1, \_\} \times \{L, R, N\} \times Q,$$

allow **multiple outputs** for a single input.

A non-deterministic Turing machine solves a problem if **there exists** a choice of transitions that leads to the correct final result.

Two definition of NP

The following are **equivalent definitions** for a decision problem being in NP:

1. Sample solutions can be checked in polynomial time.
2. A non-deterministic TM solves it in polynomial time.

Intuitively:

guessing a sample solution ≈ guessing the right transitions

Cook–Levin theorem

Theorem. SAT is NP-hard.

- Consider a non-deterministic Turing machine solving a problem in polynomial time, and run it on a sample input.
- Construct a circuit that: given a record of the Turing machine's tape/state over time, check that **(1) it followed the transition function** and **(2) ended up returning "yes"**.
- Use Tseitin transformations to turn it into a SAT formula.
- The SAT formula is satisfiable iff the original input is "yes"!

Proving NP-hardness after SAT

No more need to reason about all algorithms: use **reductions**!

A **reduction from A to B** is a way of solving A using B.

We've done:

- Reductions to graph problems
- Reductions to network flows
- Reductions to SAT

Proving NP-hardness after SAT

To show that problem A is NP-hard, we can reduce *from* SAT to A!

$$\mathbf{SAT} \leq_p A$$

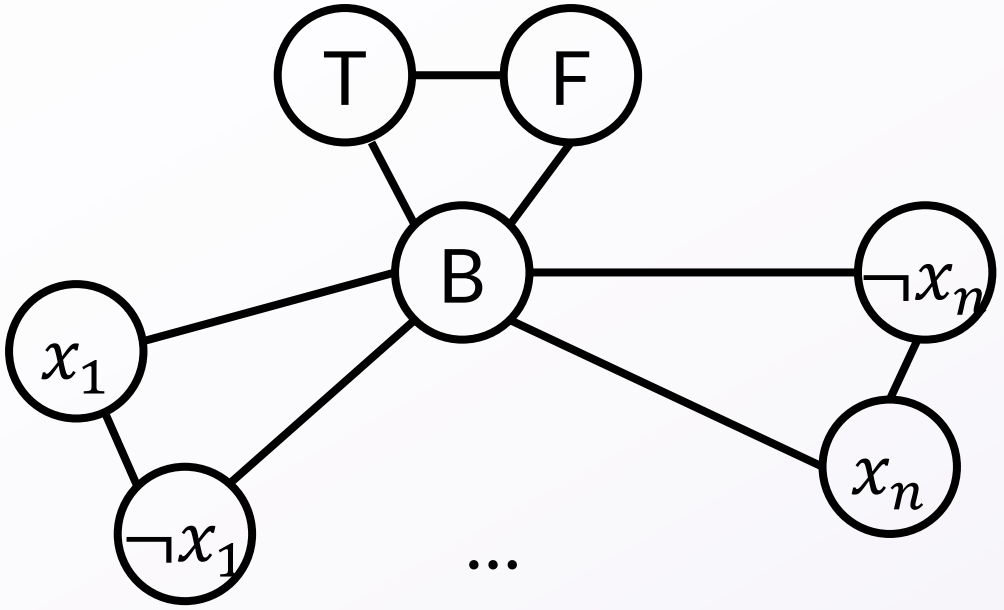
SAT is "easier" A is "harder"

Since SAT is already hard, A must be even harder! (or equally hard)

Example: 3SAT from reading

3-coloring is NP-hard

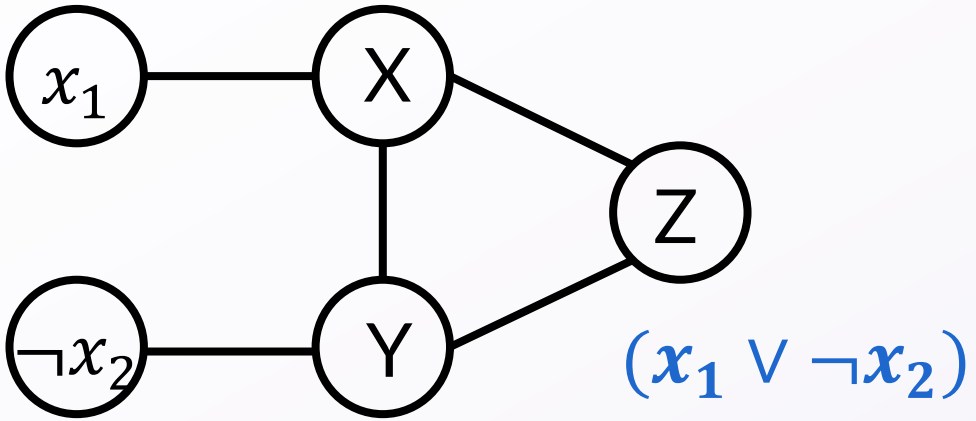
We need to solve SAT using 3-coloring.



Forces each pair **(T, F)**, $(x_1, \neg x_1), \ldots, (x_n, \neg x_n)$ to take opposite colors, neither of which is the color of B.

3-coloring is NP-hard

Imagine you had a clause $(x_1 \vee \neg x_2)$.

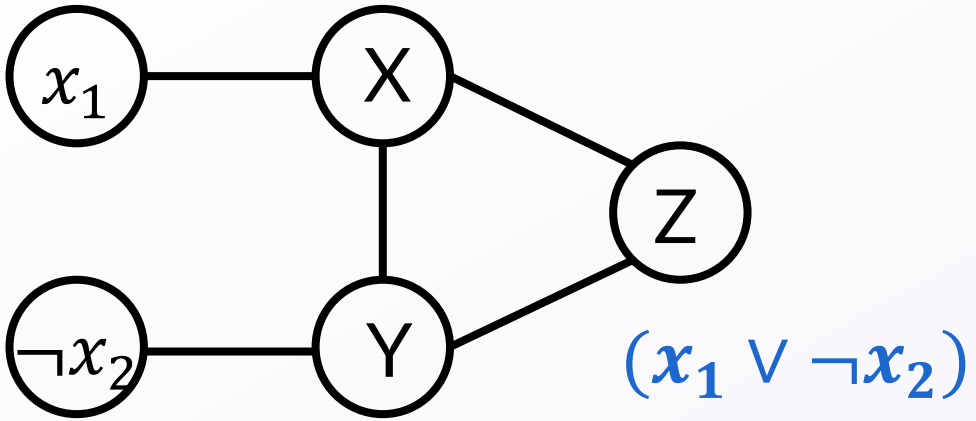


If both x_1 and $\neg x_2$ are colored F, then:

- X and Y must be different colors, both not F.
- Therefore Z must be colored F.

3-coloring is NP-hard

Imagine you had a clause $(x_1 \vee \neg x_2)$.

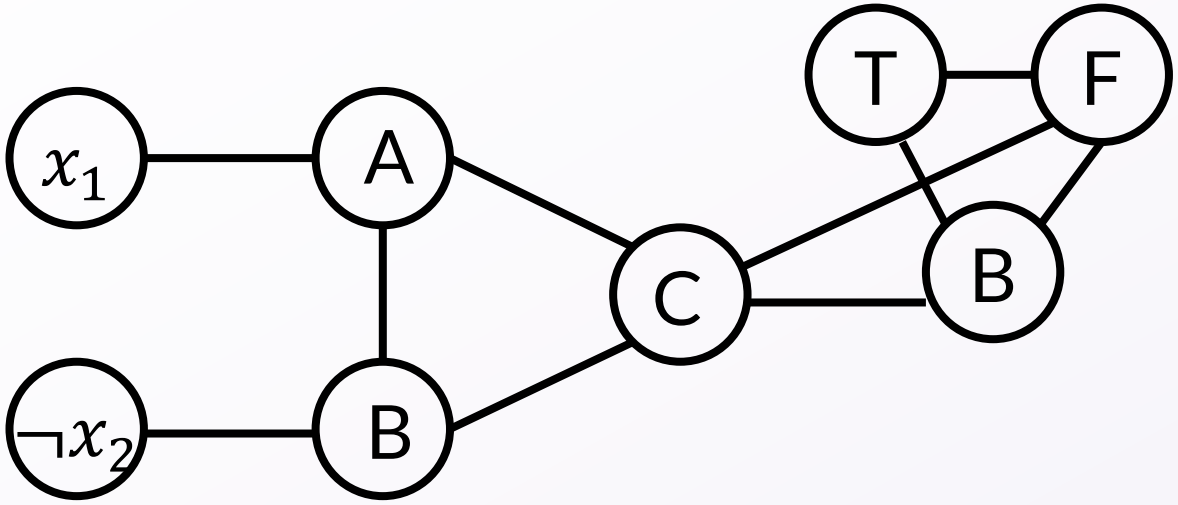


If at least one of x_1 and $\neg x_2$ are colored T, then:

- ***There exist*** colors for X and Y so that Z can be colored T.

3-coloring is NP-hard

Imagine you had a clause $(x_1 \vee \neg x_2)$.



These connections mean: C must be colored T!

The final graph after converting all clauses is 3-colorable iff the original SAT instance is satisfiable. Thus, 3-coloring is NP-complete!

Uncomputability

Some infinities are larger than others

Q: Are there more even numbers or natural numbers?

A: Both sets are infinite. Because we can pair them up, we say they are the same size:

0	1	2	3	4	5	6	7	8	9
0	2	4	6	8	10	12	14	16	18

Some infinities are larger than others

Q: Are there more real numbers or natural numbers?

A: Although both sets are infinite, there are more real numbers!

In fact, there are more real numbers just between 0 and 1 than there are natural numbers!

Some infinities are larger than others

Suppose for contradiction, I could pair up real numbers between 0 and 1 with the natural numbers.

0	0.	5	2	4	1	3	5	…
1	0.	8	2	3	7	2	3	…
2	0.	9	4	0	5	0	7	…
3	0.	6	8	0	5	1	3	…
4	0.	4	8	2	1	8	0	…
5	0.	9	4	6	2	2	4	…

The number 0.631695… is missing from this list, contradiction!

Countable vs. uncountable infinity

A set is **countably infinite** if it can be paired up with the natural numbers $\mathbb{N}$.

If it is impossible to pair with $\mathbb{N}$, then the set is **uncountable**.

$\mathbb{R}$ is an example of an uncountable set.

Infinities in computability

Q: Are there countably or uncountably many algorithms?

A: In the Java model, there are at most as many algorithms as there are finite strings (i.e. possible .java files).

Can pair all finite strings with $\mathbb{N}$:

0	1	2	3	4	5	6	7	8	9	10	…
0	1	00	01	10	11	000	001	010	011	100	…

Infinities in computability

Q: Are there countably or uncountably many problems?

A: Uncountably many, by the same "diagonal" argument!

Recall that a problem is a function $f : \{0,1\}^* \to \{0,1\}^*$.

For this argument, easier to think about having turned each binary string into a natural number.

So a problem is a function $f : \mathbb{N} \to \mathbb{N}$.

Infinities in computability

Suppose for contradiction, I could pair up all functions $f : \mathbb{N} \to \mathbb{N}$ with the natural numbers.

Inputs	0	1	2	3	4	…
f_0	56	14	0	52	4	…
f_1	68	24	78	2	34	…
f_2	119	5	46	3	10	…
f_3	24	20	25	88	4	…
f_4	7	52	15	9	8	…

The function $f(x) = f_x(x) + 1$ is missing from this list, contradiction!

Most problems are not computable!

Because there are countably many algorithms and uncountably many problems, **“most” problems are actually not computable**!

But: Luckily, most “interesting” problems are computable, as we saw throughout this course.

Halting problem

Explicit example of an uncomputable problem:

Input: A description of an algorithm (i.e. Java file) that takes all binary strings as possible input.

Goal: Does the algorithm terminate (halt) for all inputs?

You can prove this with a “diagonal argument” similar to before.

Reductions for uncomputability

Just like for NP-completeness, to show that another problem A is uncomputable,

- Identify a problem that is already known to be uncomputable (e.g. Halting problem).
- "Solve" the Halting problem assuming that you have an algorithm for A.

Because the Halting problem cannot be solved, neither can A!

Final reminders

All homework/resubmissions due Friday @ 11:59pm!

Final exam review on Friday.

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 214 if you're coming later

Nathan has online OH 12–1pm:

- https://washington.zoom.us/my/nathanbrunelle