

Lecture 21

P, NP, and computational hardness II

CSE 421 Autumn 2025

W

Previously...

When does a problem not have an efficient algorithm?

Let's back up. *Are there problems that don't have any algorithms?*

Yes! One example is the *halting problem*.

- **Input:** Program code.
- **Output:** Whether the program terminates or runs forever.

Theorem [Turing, 1936]: There is no algorithm for solving the halting problem.

We say that the halting problem is *undecidable*.

When does a problem not have an efficient algorithm?

Let's restrict to problems that are decidable, i.e. there is *some* algorithm that always outputs the correct answer. Is it necessary that those algorithms are efficient?

Are there (decision) problems that can be solved in **exponential** time, but not in **polynomial** time?

Theorem (“time hierarchy”): There exist decision problems that can be solved in *exponential* time but cannot be solved in *polynomial* time.

These are “artificial” problems. The theorem doesn't say anything about “natural” problems that we care about, e.g. Knapsack, Traveling Salesman, 3-SAT, ..

Decision problems

Definition: A *decision problem* is any problem which has a boolean (“yes” or “no”) answer.

Examples:

- Input: (G, k) . Output: Does a graph G have a vertex cover of size $\leq k$?
- Input: (G, k) . Output: Is there an MST of weight $\geq k$?
- Input: Boolean circuit φ . Output: Is there an x such that $\varphi(x) = 1$?
- Input: (W, V, \vec{w}, \vec{v}) . Output: Is there a valid Knapsack of weight $\leq W$ and value $\geq V$?
- Input: $n \times n$ Sudoku problem. Output: Is there a solution to this problem?
- Input: (G, c, s, t, k) . Output: Is there a max flow of size $\geq k$?

The classes P and EXP

Definition: P is the class of decision problems that can be solved in polynomial time, i.e. in time $t(n) = n^c$ for some constant c .

on every input



Definition: EXP is the class of decision problems that can be solved in exponential time, i.e. in time $t(n) = 2^{n^c}$ for some constant c .

Roughly speaking, we take P to be the class of (decision) problems that can be solved “efficiently”.

Example of problems in P

Some of the problems in P:

- Input: (G, u, v) . Output: Is there a path u to v of length $\leq k$?
- Input: (G, c, s, t, k) . Output: Is there a max flow of size $\geq k$?
- Input: matrices (A, B, C) . Output: If $C = A \cdot B$.
- Input: $n \in \mathbb{N}$ expressed in binary. Output: if n is prime.
- And most of the problems we saw in this class..

Today

- The class NP, reductions, and NP-completeness

The class NP

Informally: the class of problems for which “yes” inputs have an **efficiently verifiable “proof”**, and “no” inputs don’t.

The problem of “factoring” is an example.

Input: an integer x (expressed in binary), and another integer k .

Output: “yes” if x has a prime factor $\leq k$, and “no” otherwise.

If x is a “yes” instance, then, given a prime factor $\leq k$, one can efficiently verify that it divides x (even though such a factor may be hard to find).

Conversely, if x is a “no” instance, then there is no number one can give you that will (wrongly) convince you that x is a “yes” instance.

The class NP

Informally: the class of problems for which “yes” inputs have an **efficiently verifiable “proof”**, and “no” inputs don’t.

More formally:

Definition: An algorithm \mathcal{V} is a **verifier** for the problem X if, for every input x , the answer is “yes” iff there exists a proof π of length $\text{poly}(|x|)$ such that $\mathcal{V}(x, \pi) = 1$.

 A fixed polynomial for all inputs

Definition: NP is the class of decision problems that have a polynomial-time verifier.

Remark: NP stands for **non-deterministic polynomial time**.

Examples of problems in NP

Factoring

Input: an integer x (expressed in binary), and another integer k .

Output: “yes” if x has a prime factor $\leq k$, and “no” otherwise.

- Proof: π is the prime factor $\leq k$.
- **Verifier** $\mathcal{V}(x, \pi)$:
 - Check if π divides x . Output “yes” if it does, and “no” otherwise.

Can be done in polynomial time

Examples of problems in NP

Knapsack

Input: (W, V, \vec{w}, \vec{v}) .

Output: “yes” if there is a valid Knapsack of weight $\leq W$ and value $\geq V$, and “no” otherwise.

- Proof: $\pi \in \{0,1\}^n$ such that $\pi_i = 1$ if we should include item i .
- **Verifier** $\mathcal{V}(x = (W, V, \vec{w}, \vec{v}), \pi)$:
 - Check if $\sum_{i:\pi_i=1} w_i \leq W$ and $\sum_{i:\pi_i=1} v_i \geq V$. Output “yes” if both conditions hold, and “no” otherwise.

↗ Can be done in polynomial time

Examples of problems in NP

k -coloring

Input: A graph G .

Output: “yes” if there is a valid k -coloring, and “no” otherwise.

- Proof: π is the coloring itself (i.e. an assignment of a color to each vertex)
- **Verifier** $\mathcal{V}(G, \pi)$:
 - Check that π is a valid coloring. If so, output “yes”. Otherwise “no”.

Can be done in polynomial time

Examples of problems in NP

3SAT

Input: A 3-CNF formula φ .

Output: “yes” if there exists a $z \in \{0,1\}^n$ such that $\varphi(z) = 1$. “No” otherwise.

A 3-CNF is a function $\varphi : \{0,1\}^n \rightarrow \{0,1\}$ defined as an AND of ORs, where each OR acts on at most 3 variables (i.e. bits of the input) or their negations

e.g. $\varphi(z) = (z_1 \vee z_3) \wedge (z_2 \vee z_6 \vee \bar{z}_7) \wedge (\bar{z}_1 \vee z_5)$

- Proof: Let π simply be z such that $\varphi(z) = 1$.
- **Verifier** $\mathcal{V}(\varphi, \pi)$:
 - Check that $\varphi(\pi) = 1$. If so, output “yes”. Otherwise “no”.



Can be done in polynomial time

Examples of problems in NP

Min Cut

Input: (G, c, s, t, k) .

Output: “Yes” if there is an s-t cut of size $\leq k$, “no” otherwise.

- Proof: $\pi \in \{0,1\}^{|V|}$ describes the vertices in S for an s-t cut (S, T)
- **Verifier** $\mathcal{V}(x, \pi)$:
 - Check that π describes a valid s-t cut.
 - Compute, $c(S, T) = \sum_{(u,v) \in E : \pi_u=1, \pi_v=0} c(u, v)$
 - Check if $c(S, T) \leq k$

Can be done in polynomial time

*Can you think of a different verifier
(and proof) for this problem?*

Examples of problems in NP

Min Cut

Input: (G, c, s, t, k) .

Output: “Yes” if there is an s-t cut of size $\leq k$, “no” otherwise.

- Proof: empty string
- **Verifier** $\mathcal{V}(x)$:
 - Compute a min cut (S, T) using Edmonds-Karp flow algorithm.
 - Compute, $c(S, T) = \sum_{(u,v) \in E : u \in S, v \in T} c(u, v)$
 - Check if $c(S, T) \leq k$.

Any problem in P is also in NP:
the verifier can ignore the proof,
and solve the problem by itself.

P, NP, and EXP

- P : decision problems with a poly-time algorithm
- NP : decision problems with a poly-time verifier
- EXP : decision problems with a exp-time algorithm

Theorem: $P \subseteq NP$

by the previous slide

Theorem: $NP \subseteq EXP$

Proof:

P, NP, and EXP

- P : decision problems with a poly-time algorithm
- NP : decision problems with a poly-time verifier
- EXP : decision problems with a exp-time algorithm

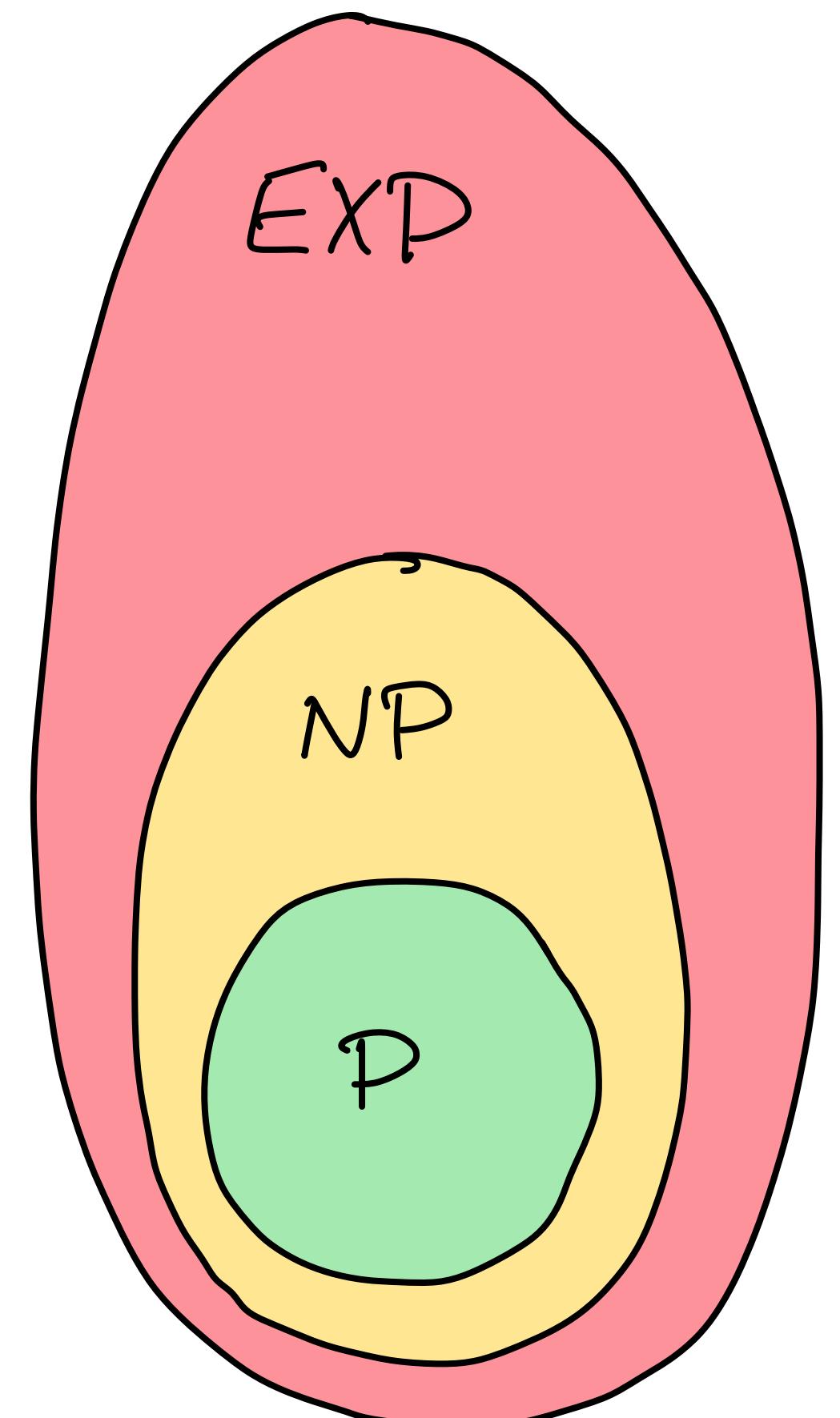
Theorem: $P \subseteq NP$

by the previous slide

Theorem: $NP \subseteq EXP$

Proof: If a problem is in NP, then one can always do a brute force search for a proof that is accepted by the verifier. Since the proof is required to be of length $\leq \text{poly}(|x|)$, this search can be done in time $2^{\text{poly}(|x|)}$.

We don't know if inclusions are strict..



The million dollar question: Is $P \stackrel{?}{=} NP$?

- Morally, the question asks: is the problem of **deciding** if something is true as easy as **verifying** a proof that it is true?
- There is a \$1 million bounty for solving the problem (in either direction!)
- If yes: There is a poly-time algorithm for every NP problem
- If no: No efficient/poly-time algorithm for some problems such as 3-coloring, Traveling Salesman, 3SAT, Knapsack, Vertex Cover..

NP-completeness

Definition: A problem is NP-complete if (a) it is in NP and (b) it is the “hardest” problem in NP.

What does it mean for problem Y to be harder than (or at least as hard as) problem X ?

Informally: it means that a subroutine to solve Y can be used to efficiently solve X

NP-completeness

What does it mean for problem Y to be harder than (or, at least as hard as) problem X ?

Informally: it means that a subroutine to solve Y can be used to efficiently solve X .

This is formalized by the notion of a **reduction**.

Definition: We say that X **reduces** to Y , denoted $X \leq_p Y$, if there exists an algorithm R that maps instances of X to instances of Y such that:

- R is polynomial-time
 - x is a “yes” instance of X if and only if $y = R(x)$ is a “yes” instance of Y

Note: If $X \leq_p Y$, then we can solve any instance x of X as follows:

- Compute $y = R(x)$, an instance of Y .
 - Run a subroutine to solve y .

In other words, up to the polynomial-time overhead of R (which we consider “efficient”), we can solve X in the same time it takes to solve Y .

NP-completeness

We can now formalize the definition of NP completeness.

Definition: A problem is NP-complete if (a) it is in NP and (b) it is the “hardest” problem in NP.

NP-completeness

We can now formalize the definition of NP completeness.

Definition: A problem Y is NP-complete if

- (a) $Y \in \text{NP}$
- (b) For every problem $X \in \text{NP}$, it holds that $X \leq_p Y$

Condition (b) on its own is known as “NP-hardness”.

We've seen reductions before...

- We've seen reductions many times before in this class
- Anytime you used an algorithm as a subroutine – you were performing a reduction
- Examples:
 - From stable matching with *arbitrary* numbers of proposers and receivers to stable matching with *equal* number of proposers and receivers
 - From bipartite matching to Max Flow
 - From shortest path picking up k items to standard shortest path
 - Etc..

Example of a reduction

Subset Sum \leq_p Decision-Knapsack

- Subset Sum: Given input a_1, \dots, a_n, T , decide if there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} a_i = T$.
- Decision-Knapsack: Given input $w_1, \dots, w_n, v_1, \dots, v_n, W, V$, decide if there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq V$.
- **Reduction:** We want to come up with an algorithm \mathcal{A}' for solving Subset Sum from an algorithm \mathcal{A} for solving Knapsack. \mathcal{A}' does the following:
 - Given input a_1, \dots, a_n, T , define $w_i = v_i \leftarrow a_i$ and $W = V \leftarrow T$.
 - Then run \mathcal{A} on $(w_1, \dots, w_n, v_1, \dots, v_n, W, V)$.

Proving a reduction is correct

Recall: a reduction $X \leq_p Y$ between two decision problems X and Y is a poly-time algorithm R such that:

- “yes” \rightarrow “yes”: If $x \in X$ is a “yes” instance, then $R(x) \in Y$ is also a “yes” instance
- “no” \rightarrow “no”: If $x \in X$ is a “no” instance, then $R(x) \in Y$ is also a “no” instance

Crucially, you have to show both implications!

Example of a reduction

Subset Sum \leq_p Decision-Knapsack

Proof of correctness of the reduction:

- $x = (\vec{a}, T) \in \text{Subset Sum}$, $R(x) = (\vec{w} = \vec{v} \leftarrow \vec{a}, W = V \leftarrow T)$
- If x is a “yes” instance, then there exists $S \subseteq [n]$ s.t. $\sum_{i \in S} a_i = T$
 - Therefore, $\sum_{i \in S} w_i = \sum_{i \in S} a_i = T \leq W$,
 $\sum_{i \in S} v_i = \sum_{i \in S} a_i = T \geq V$. So $R(x)$ is a “yes” instance.
- If x is a “no” instance, then for all $S \subseteq [n]$, $\sum_{i \in S} a_i \neq T$.
 - Suppose for a contradiction that $R(x) = (\vec{w} = \vec{v} \leftarrow \vec{a}, W = V \leftarrow T)$ was a “yes” instance
 - Then, there is $S \subseteq [n]$ such that $\sum_{i \in S} w_i = \sum_{i \in S} a_i = \leq T$,
 $\sum_{i \in S} v_i = \sum_{i \in S} a_i = \geq T$, i.e. $\sum_{i \in S} a_i = T$. This is a contradiction.

Consequences of NP-completeness

Definition: A problem Y is NP-complete if

- (a) $Y \in \text{NP}$
- (b) For every problem $X \in \text{NP}$, it holds that $X \leq_p Y$

Theorem: Let Y be a NP-complete problem. Then Y is solvable in poly-time iff $\text{P} = \text{NP}$.

Proof:

- (\Leftarrow) If $\text{P} = \text{NP}$, then Y has a poly-time algorithm since $Y \in \text{NP} = \text{P}$
- (\Rightarrow) Let X be any problem in NP . Since $X \leq_p Y$, we can solve X in poly-time using the poly-time algorithm for Y as a subroutine. So $X \in \text{P}$. So $\text{P} = \text{NP}$.

Fundamental question: Do there exist “natural” NP-complete problems?

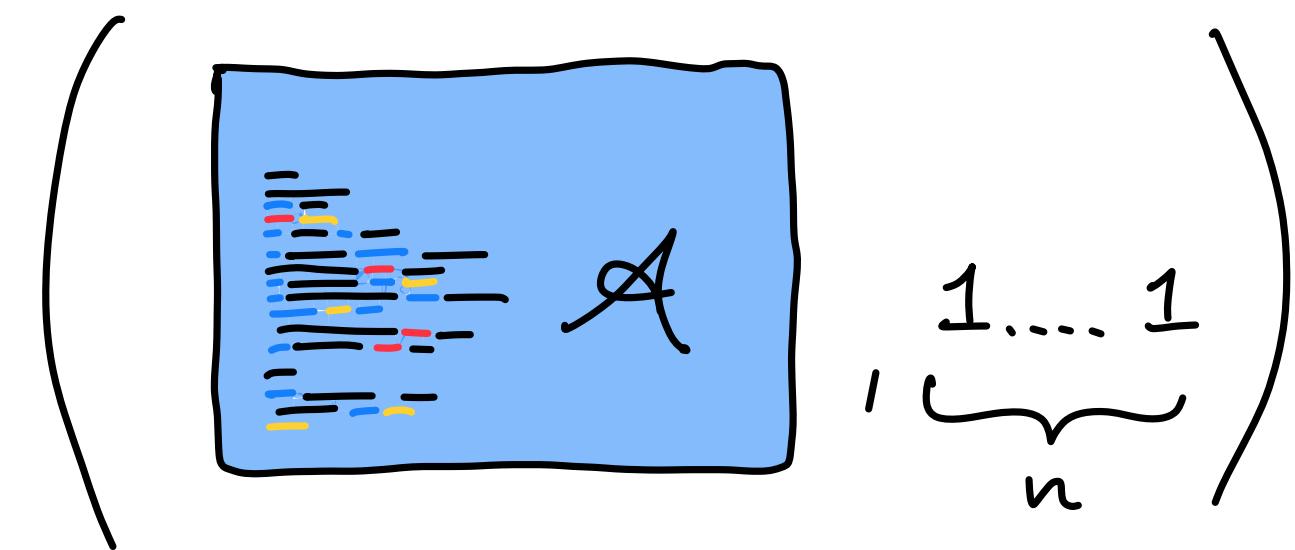
A partial list of NP-complete problems

- Boolean function satisfiability, 3-SAT
- Graph problems: Vertex Cover, 3-coloring, Independent Set, Max Cut
- Path and cycle problems: Hamiltonian path, Traveling Salesman
- Combinatorial optimization problems: Knapsack, Subset Sum
- 0-1 Integer Programming

The “first” NP-complete problem

Satisfiability

input:



Input: $(\langle \mathcal{A} \rangle, n)$, the description of a polynomial-time algorithm \mathcal{A} and integer n in unary.

Output: Whether there exists a π such that $\mathcal{A}(\pi) = 1$ and $|\pi| = n$.

Theorem: Satisfiability is NP-complete.

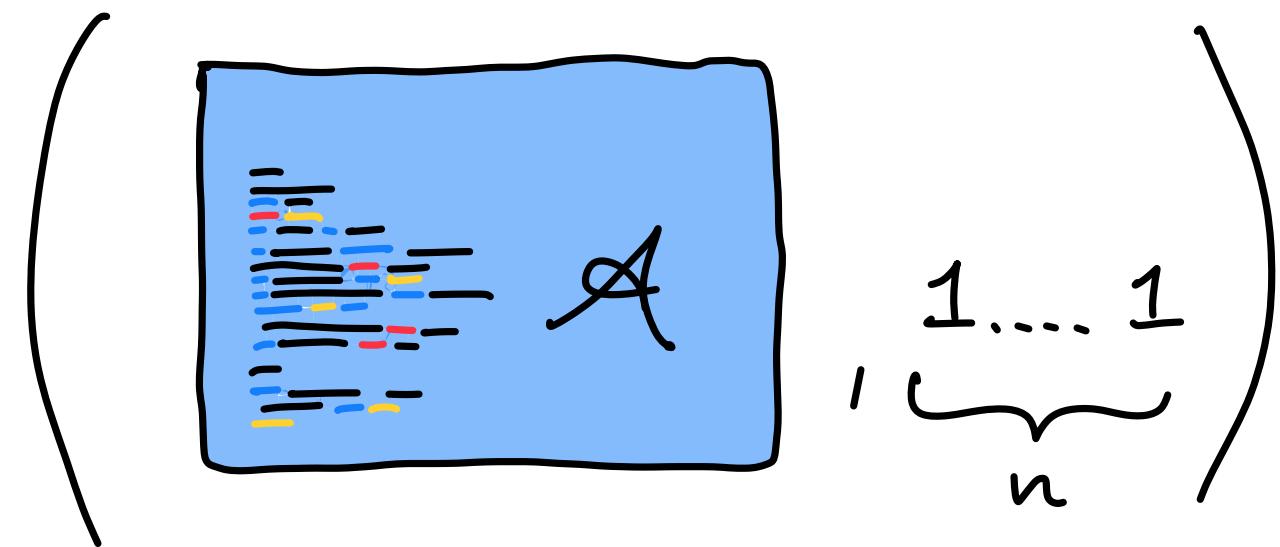
Proof: *Satisfiability is “trivially” in NP*: for a “yes” instance $(\langle \mathcal{A} \rangle, n)$, take the proof to be π such that $|\pi| = n$ and $\mathcal{A}(\pi) = 1$. The verifier simply checks that this is the case.

By definition of satisfiability, for a “no” instance, such a π doesn’t exist.

The “first” NP-complete problem

Satisfiability

input:



Input: $(\langle \mathcal{A} \rangle, n)$, the description of a polynomial-time algorithm \mathcal{A} and integer n in unary.

Output: Whether there exists a π such that $\mathcal{A}(\pi) = 1$ and $|\pi| = n$.

Theorem: Satisfiability is NP-complete.

Proof: NP-hardness: let X be any other problem in NP. Our goal is to show $X \leq_p Y$.

Since $X \in \text{NP}$, there is poly-time verifier \mathcal{V} satisfying the properties in the definition of NP. Let $p(n)$ denote the length of a proof for an instance of size n . Define the reduction algorithm R so that:

$$x \in X \quad \mapsto \quad (\langle \mathcal{A} \rangle, n) := (\langle \mathcal{V}_x \rangle, p(|x|))$$

where \mathcal{V}_x is shorthand for $\mathcal{V}(x, \cdot)$.

- “Yes” maps to “Yes”: If $x \in X$ is a “yes” instance, there exists π of length $p(|x|)$ s.t. $\mathcal{V}_x(\pi) = 1$. This means precisely that $(\langle \mathcal{V}_x \rangle, p(|x|))$ is a “yes” instance of satisfiability.
- “No” maps to “No”: If $x \in X$ is a “no” instance, then for all π of length $p(|x|)$, $\mathcal{V}_x(\pi) = 0$. This means precisely that $(\langle \mathcal{V}_x \rangle, p(|x|))$ is a “no” instance of satisfiability.

Proving more problems are NP-complete

Recipe for showing that a problem Y is NP-complete

- Step 1: Show that $Y \in \text{NP}$.
- Step 2: Choose a known NP-complete problem X .
- Step 3: Prove that $X \leq_p Y$.

Correctness of recipe:

- First, notice that \leq_p is a transitive operation, i.e. if $W \leq_p X$ and $X \leq_p Y$ then $W \leq_p Y$.
- Now, suppose you have shown Steps 1,2,3. Consider any $W \in \text{NP}$. We have $W \leq_p X$, since X is NP-complete. The latter combined with step 3 (and transitivity) implies $W \leq_p Y$. So, Y is NP-complete.

Alternatively, you can skip Step 2, and instead show that Step 3 holds for *any* problem $X \in \text{NP}$.

3-SAT problem

- The 3-SAT problem is the most well known of all NP-complete problems
- A boolean formula φ is a 3-SAT formula over variables $x_1, \dots, x_n \in \{0,1\}$ if
 - $\varphi = \varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_k$, the “AND” of k subformulas
 - Each φ_j is the “OR” of ≤ 3 variables or their negations from x_1, \dots, x_n .
 - Examples: e.g. $\varphi(x) = (x_1 \vee x_3) \wedge (x_2 \vee x_6 \vee \bar{x}_7) \wedge (\bar{x}_1 \vee x_5 \vee x_6)$

Theorem: 3-SAT is NP-complete.

Proof:

3-SAT problem

- The 3-SAT problem is the most well known of all NP-complete problems
- A boolean formula φ is a 3-SAT formula over variables $x_1, \dots, x_n \in \{0,1\}$ if
 - $\varphi = \varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_k$, the “AND” of k subformulas
 - Each φ_j is the “OR” of ≤ 3 variables or their negations from x_1, \dots, x_n .
 - Examples: e.g. $\varphi(x) = (x_1 \vee x_3) \wedge (x_2 \vee x_6 \vee \bar{x}_7) \wedge (\bar{x}_1 \vee x_5 \vee x_6)$

Theorem: 3-SAT is NP-complete.

Proof: next time.