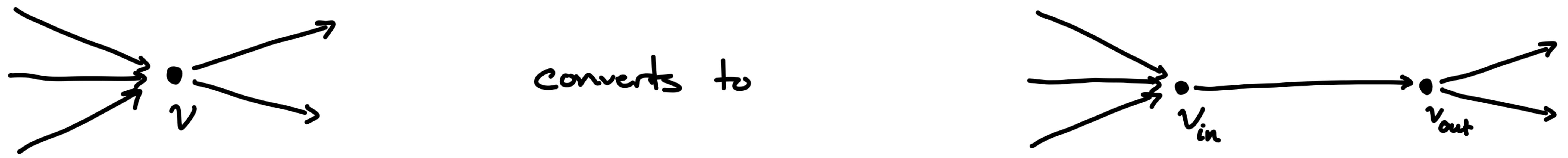
Lecture 20

P and NP

Chinmay Nirkhe | CSE 421 Winter 2026

General max flow/min cut algorithmic paradigm

- If source and sink are not obvious, they may need to be added to the graph
- We need to choose capacity limits for edges: 1, ∞ or an input from the problem are logical choices
    - Edges of capacity ∞ **cannot** cross the cut. Equivalently, edges of free flow.
- Undirected graphs will need to be converted to directed equivalents
    - Unnecessary flow cycles can be removed after flow is calculated
- Split a vertex into two (will show up on problem set):



- Choose correct version of flow algorithm based on capacities

Cut like problems

- Until now, most of the problems looked mostly “flow”-like
- Max flow = min cut tells us that there are probably many “cut”-like problems we can also solve
- Next: an examples of a cut-like problem
  - Goal here is to get you to see flow networks appear in unexpected situations
  - This is at the heart of learning how to design algorithms

Baseball winner

- Imagine a simplified scenario — the team(s) that wins the most games overall is crowned the winner(s).
- Midway through the season, we have the following win totals for the teams

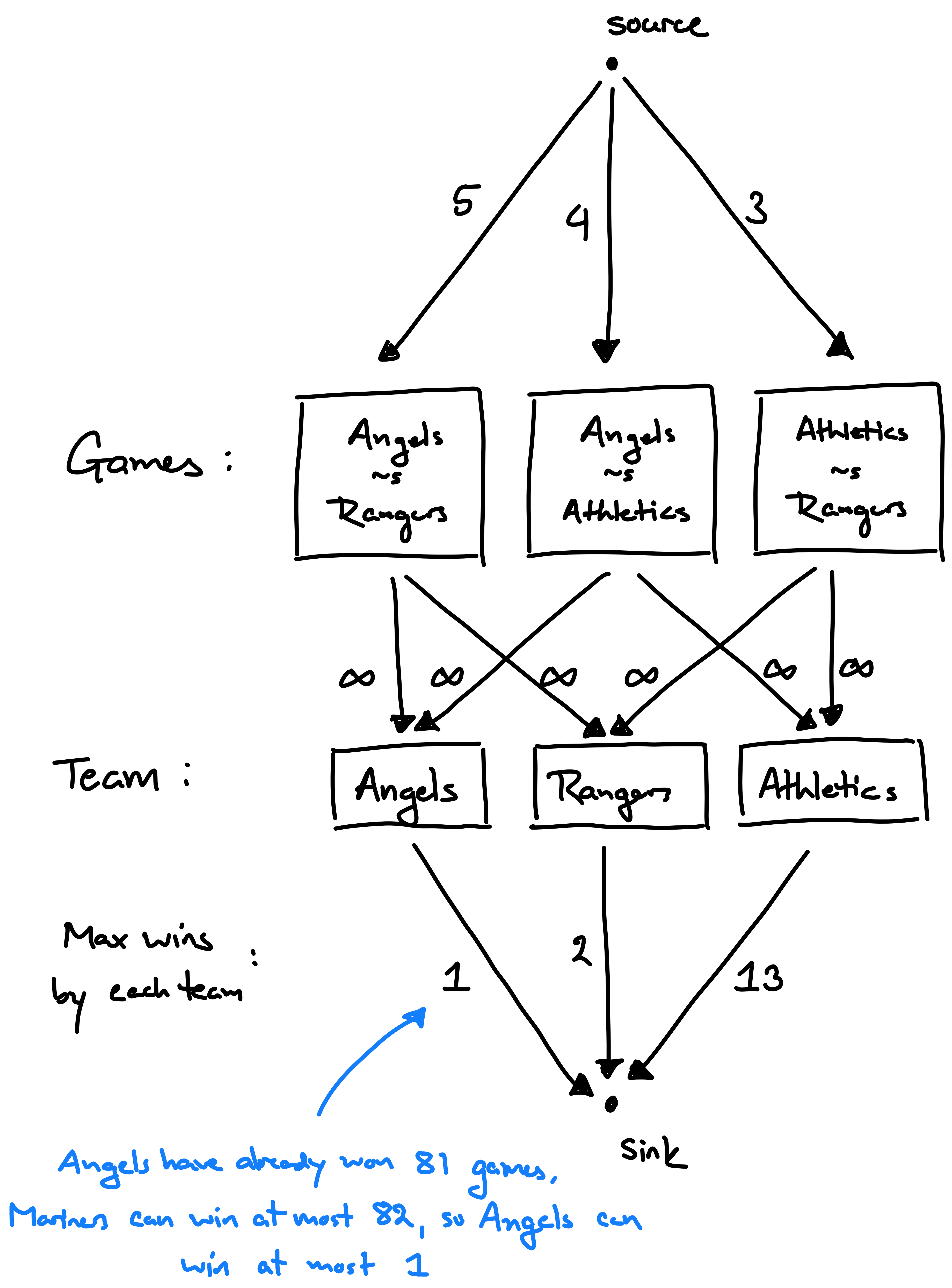
Team	Wins	Games remaining vs Angels	Games remaining vs Rangers	Games remaining vs Athletics	Games remaining vs Mariners
Angels	81	—	5	4	3
Rangers	80	5	—	3	4
Athletics	69	4	3	—	5
Mariners	70	3	4	5	—

Could the Mariners possibly win or tie for first?

Baseball winner

- Best case is Mariners win out — 82 wins
- Still depends on how the other teams play each other. How do we algorithmically calculate this?
- In order to win or tie, Mariners must have a run total at least as high as every other team.

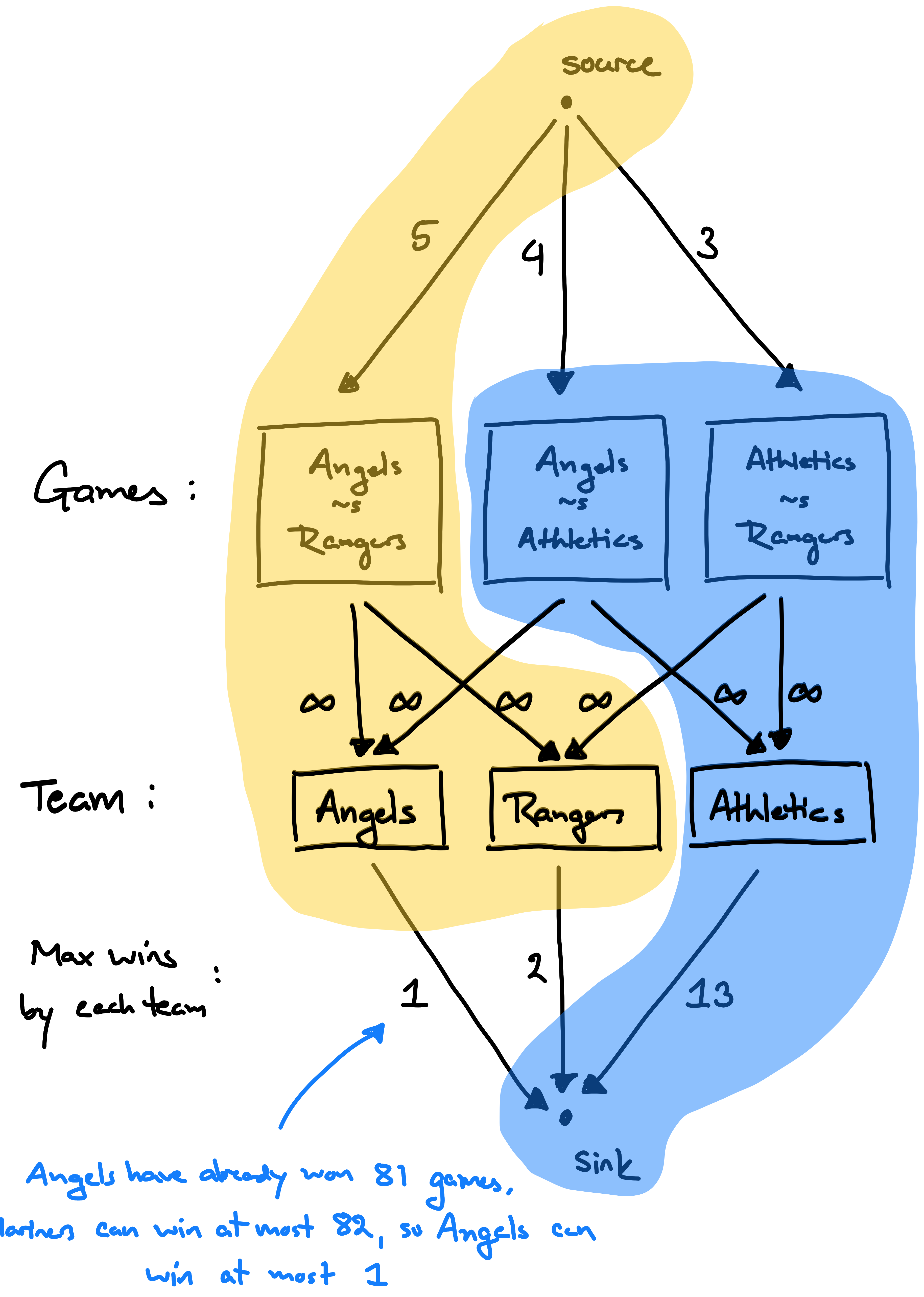
Team	Wins	Games remaining vs Angels	Games remaining vs Rangers	Games remaining vs Athletics	Games remaining vs Mariners
Angels	81	—	5	4	3
Rangers	80	5	—	3	4
Athletics	69	4	3	—	5
Mariners	70	3	4	5	—



Baseball winner

- If there was a way that the games could play out such that no team amassed > 82 wins then there would be a flow of value $5 + 4 + 3 = 12$ in this network.
- However, the min cut equals = 10

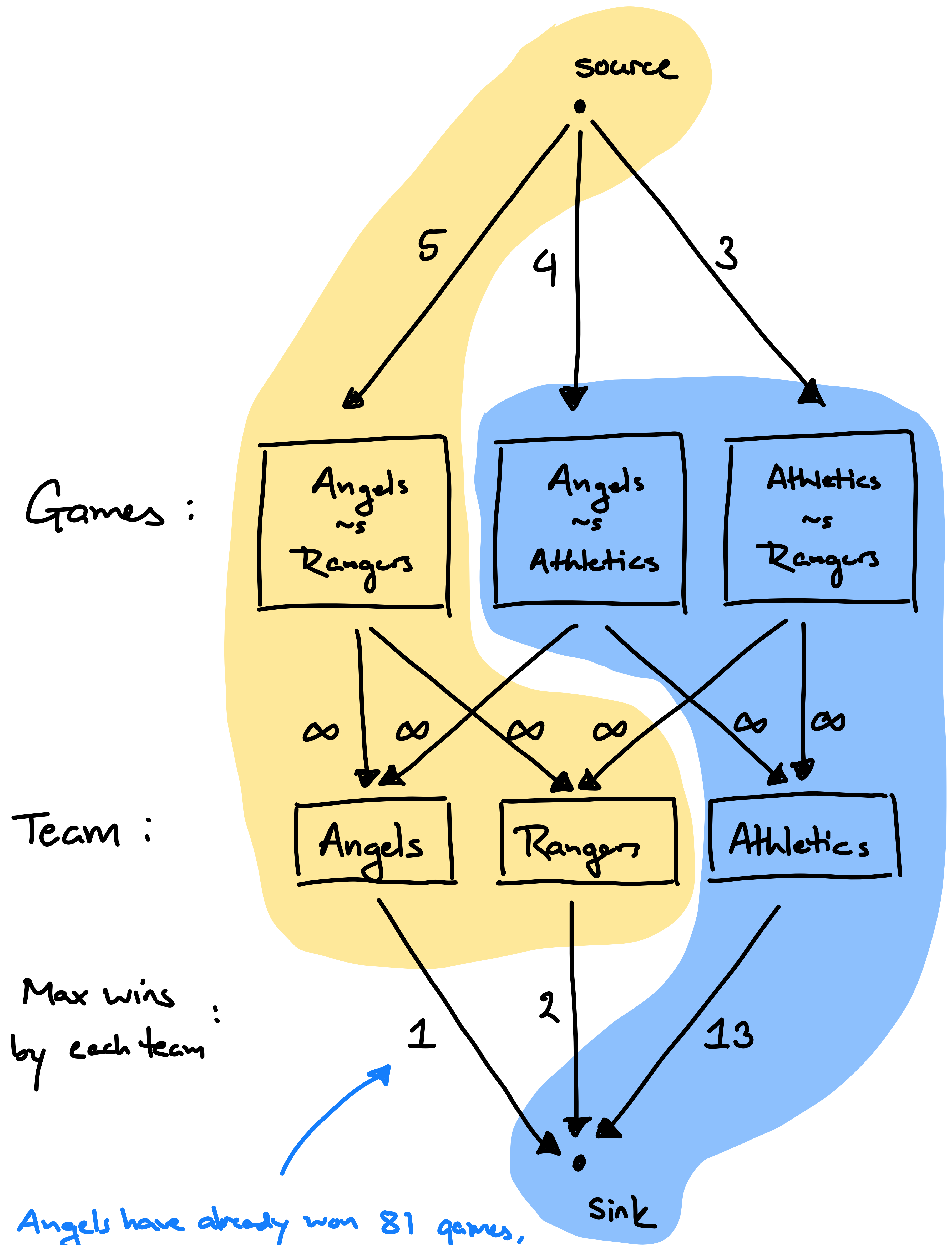
Team	Wins	Games remaining vs Angels	Games remaining vs Rangers	Games remaining vs Athletics	Games remaining vs Mariners
Angels	81	—	5	4	3
Rangers	80	5	—	3	4
Athletics	69	4	3	—	5
Mariners	70	3	4	5	—



Baseball winner

- Even though no team has won > 82 games yet, this mathematically proves that the Mariners cannot win/tie for 1st.
- A clever way to consider all possible scenarios without exploring all the remaining games.

Team	Wins	Games remaining vs Angels	Games remaining vs Rangers	Games remaining vs Athletics	Games remaining vs Mariners
Angels	81	—	5	4	3
Rangers	80	5	—	3	4
Athletics	69	4	3	—	5
Mariners	70	3	4	5	—



Angels have already won 81 games, Mariners can win at most 82, so Angels can win at most 1

P vs NP

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 called the *halting problem*.
  - **Input:** Program code.
  - **Output:** Whether the program code every terminates or runs forever.
- **Theorem** [Gödel]**:** There is no algorithm for solving the halting problem.
- **Theorem:** Solving a system of polynomial equations for integer solutions has no algorithm.

When does a problem not have an efficient algorithm?

- Let’s restrict to problems that have algorithms. Is it necessary that those algorithms are efficient?
- **Theorem:** There exist problems that can be solved in exponential time but cannot be solved in polynomial time.
    - This theorem just proves the existence of such problems — it does not prove that there are “interesting problems” that require exponential time.
    - Interesting problems like: Vertex Cover, Independent Set, Knapsack problem, Traveling Salesman, 3-Color, etc. What about those?

Decision problems

- **Definition:** A *decision problem* is any problem which has a boolean (yes vs. no) answer.
  - 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 class P

- **Definition:** An algorithm $\mathscr{A}$ runs in time $t(n)$ if for every input x, $\mathscr{A}(x)$ terminates in at most $\leq t(|x|)$ "steps".
  - An algorithm runs in **polynomial-time** (poly-time) if $t(n) = n^c$ for some constant c .
  - We say a decision problem can be solved in polynomial-time if there is a polynomial-time algorithm $\mathscr{A}$ for it
- **Definition:** The class P is the class of decision problems that can be solved in polynomial-time
  - P is a decent approximation for the set of problems that can be solved efficiently by some model of computation. In practice, we are interested in the problems with poly runtimes for some constants c.

The class P

- Some of the problems in P
  - Input: (G, k). Output: Is there an MST of weight $\geq k$?
  - Input: (G, c, s, t, k). Output: Is there a max flow of size $\geq k$?
  - Input: (A, b, c). Output: Value of LP $\max c^\top x \text{ s.t. } Ax \leq b, x \geq 0$
  - Input: matrices (A, B, C). Output: If $C = A \cdot B$.
  - Input: $n \in \mathbb{N}$ expressed in binary. Output: if n is prime.

The class NP

- Certification algorithm intuition: A certifier algorithm doesn't determine whether the answer to a decision problem is "yes" on its own. Rather, it checks a proof (a.k.a. certificate a.k.a. witness) π that the answer is "yes".

- **Definition:** An algorithm $\mathscr{V}(x, \pi)$ is a **certifier/verifier** for the problem X if for every string x, the answer is "yes" IFF there exists a proof π such that $\mathscr{V}(x, \pi) = 1$.

    - A certifier is poly-time if $|\pi| \leq |x|^{c'}$ and $\mathscr{V}$ runs in time $|x|^c$ for some constants c, c'.

- **Definition:** The class NP is the class of decision problems for which there is a poly-time certifier.

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

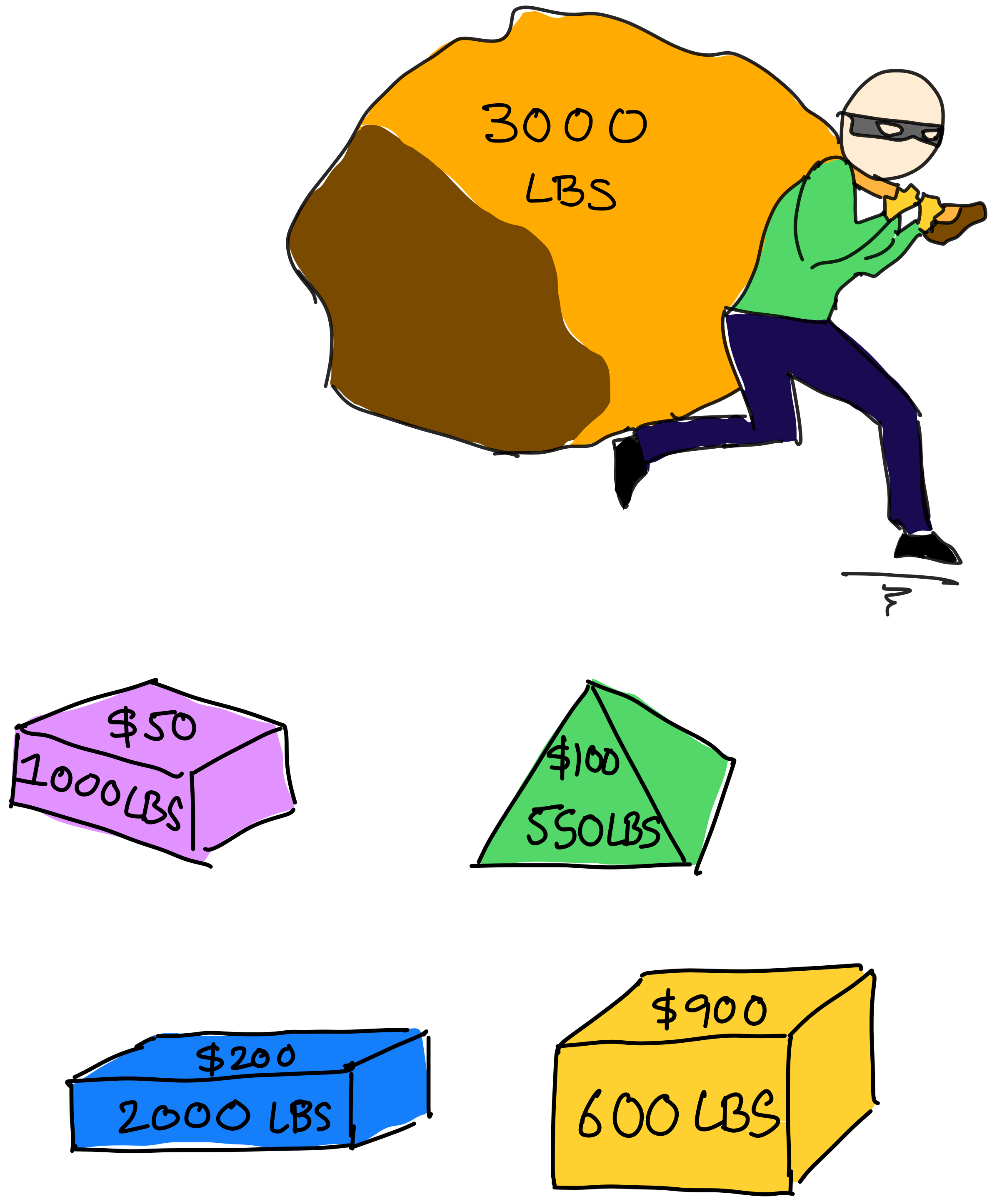
Examples of problems in NP

Knapsack

- Input: $(W, V, \overrightarrow{w}, \vec{v})$. Output: Is there a valid Knapsack of weight $\leq W$ and value $\geq V$?
- Proof: $\pi \in \{0,1\}^n$. $\pi_i = 1$ if we should include item i.
- **Certifier algorithm** $\mathcal{V}\left(x = (W, V, \overrightarrow{w}, \vec{v}), \pi\right)$:
  - Test if $\sum_{i:\pi_i=1} w_i \leq W$ and $\sum_{i:\pi_i=1} v_i \geq V$
    - Respond "yes" if both conditions hold
    - Otherwise, respond "no".

Examples of problems in NP
Knapsack

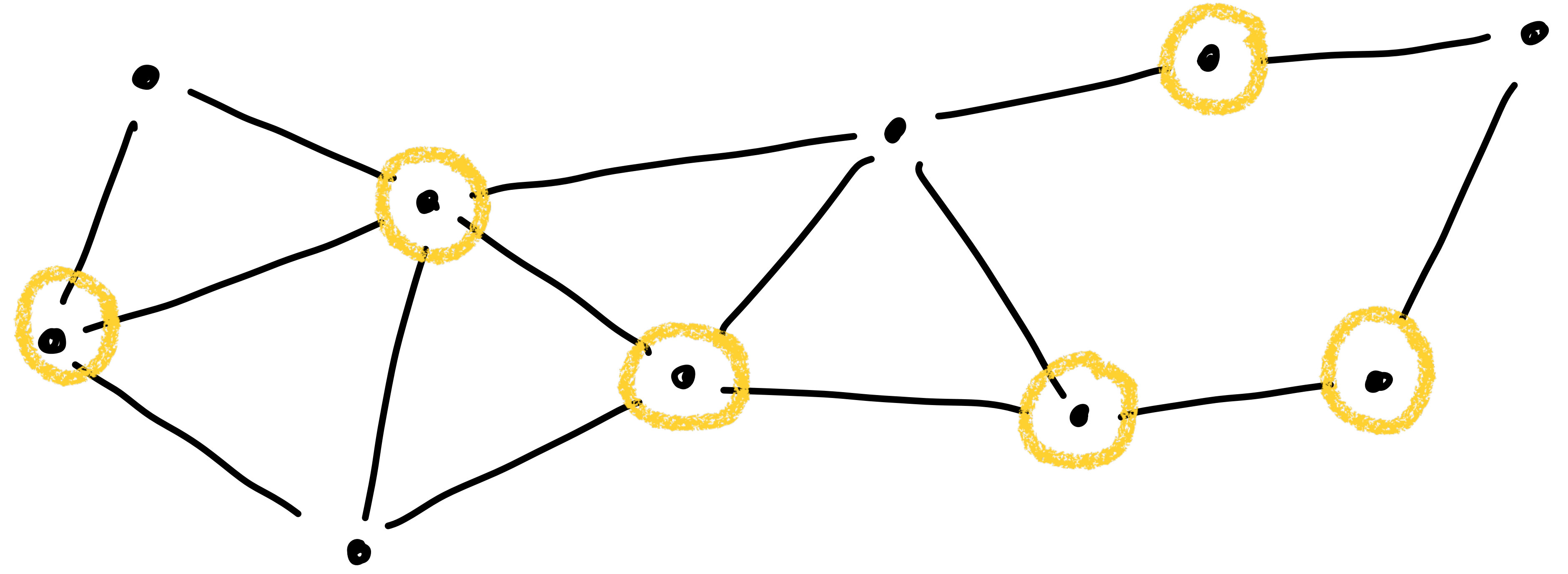
- Input: $(W, V, \vec{w}, \vec{v})$. Output: Is there a valid Knapsack of weight $\leq W$ and value $\geq V$?
- Proof: $\pi \in \{0,1\}^n$. $\pi_i = 1$if we should include item i.
- **Correctness**:
  - If there is a valid Knapsack,
    - let $S \subseteq [n]$ be the items. Set $\pi_i = 1$ iff $i \in S$.
    - Then there exists a π s.t. $\mathscr{V}$ will accept.
  - If there exists a proof π which is accepted by $\mathscr{V}$,
    - then $S \subseteq [n]$ the items i s.t. $\pi_i = 1$ is a valid Knapsack.
  - Bijection between Knapsacks and proofs.



Examples of problems in NP

Vertex Cover

- Input: (G, k). Output: Does a graph G have a vertex cover of size $\leq k$?
- Proof: $\pi \in \{0,1\}^V$. $\pi_v = 1$if we should include vertex v in the cover.
- **Certifier algorithm** $\mathscr{V}\left(x = (G, k), \pi\right)$:
  - For all edges $e = (u, v) \in E$, test that $\pi_u = 1$ or $\pi_v = 1$.
  - Test that $\sum_{v \in V} \pi_v \leq k$.



vertex cover of size 6

Examples of problems in NP
3SAT

- Input: 3-CNF formula φ. Output: If there exists a $z \in \{0,1\}^n$ such that $\varphi(z) = 1$.

3-CNF is a boolean fn defined as a AND of ORs where each OR acts on ≤ 3 variables or their negations.

Ex. Single clause $\varphi(z_1, z_2, z_3) = (z_1 \vee \neg z_2 \vee z_3)$

Double clause $\varphi(z_1, \dots, z_4) = (z_1 \vee \neg z_2 \vee z_3) \wedge (\neg z_1 \vee \neg z_2 \vee z_4)$

In general $\varphi(z_1, \dots, z_n) = (\_ \vee \_ \vee \_) \wedge \dots \wedge (\_ \vee \_ \vee \_)$

Examples of problems in NP

3SAT

- Input: 3-CNF formula φ. Output: If there exists a $z \in \{0,1\}^n$ such that $\varphi(z) = 1$.

$$\varphi(z_1, \dots, z_4) = (z_1 \vee \neg z_2 \vee z_3) \wedge (\neg z_1 \vee \neg z_2 \vee z_4)$$

$$\varphi(0, 1, 0, 0) = (0 \vee \neg 1 \vee 0) \wedge (\neg 0 \vee \neg 1 \vee 0)$$

$$= (0 \vee 0 \vee 0) \wedge (1 \vee 0 \vee 0)$$

$$= 0 \wedge 1$$

$$= 0$$

Examples of problems in NP

3SAT

- Input: 3-CNF formula φ. Output: If there exists a $z \in \{0,1\}^n$ such that $\varphi(z) = 1$.

$$\varphi(z_1, \ldots, z_4) = (z_1 \vee \neg z_2 \vee z_3) \wedge (\neg z_1 \vee \neg z_2 \vee z_4)$$

$$\varphi(1,1,0,1) = (1 \vee \neg 1 \vee 0) \wedge (\neg 1 \vee \neg 1 \vee 1)$$

$$= (1 \vee 0 \vee 0) \wedge (0 \vee 0 \vee 1)$$

$$= 1 \wedge 1$$

$$= 1$$

Examples of problems in NP

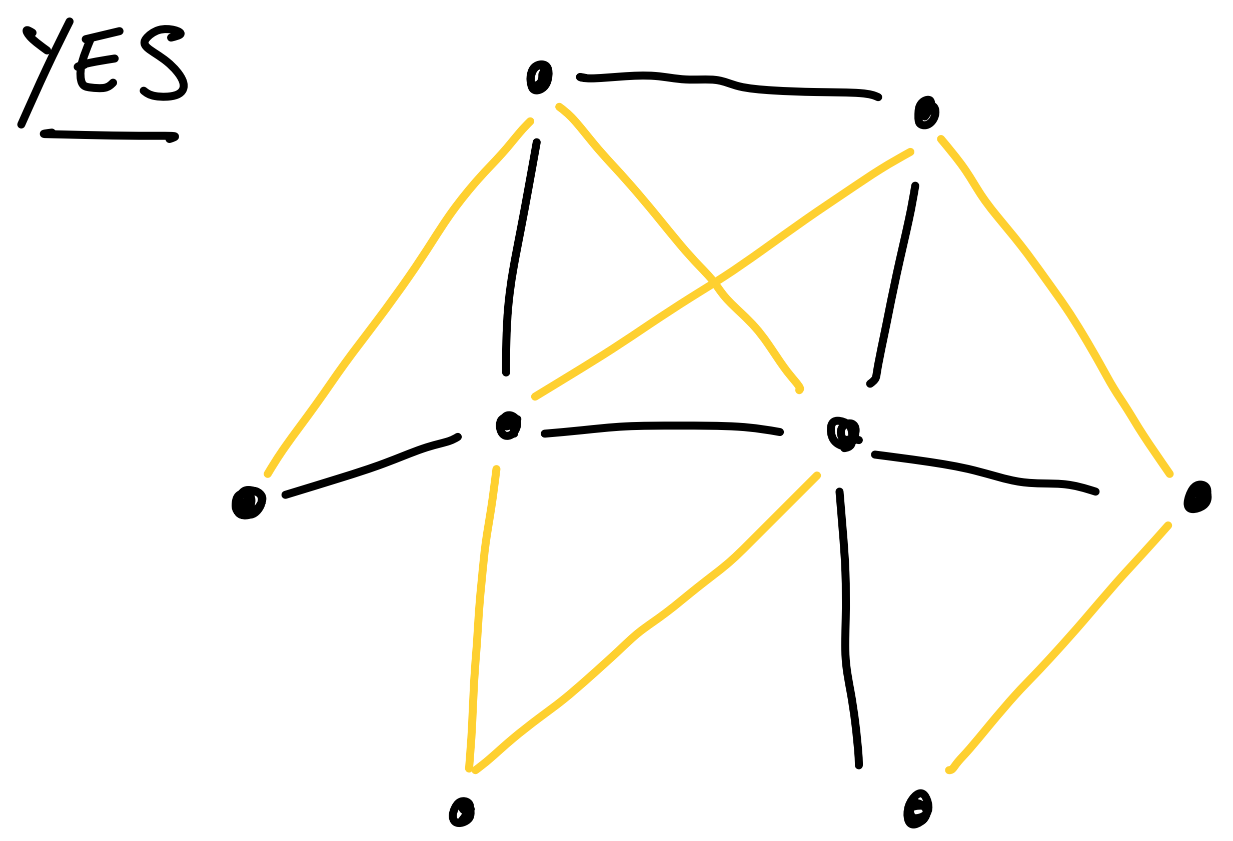
3SAT

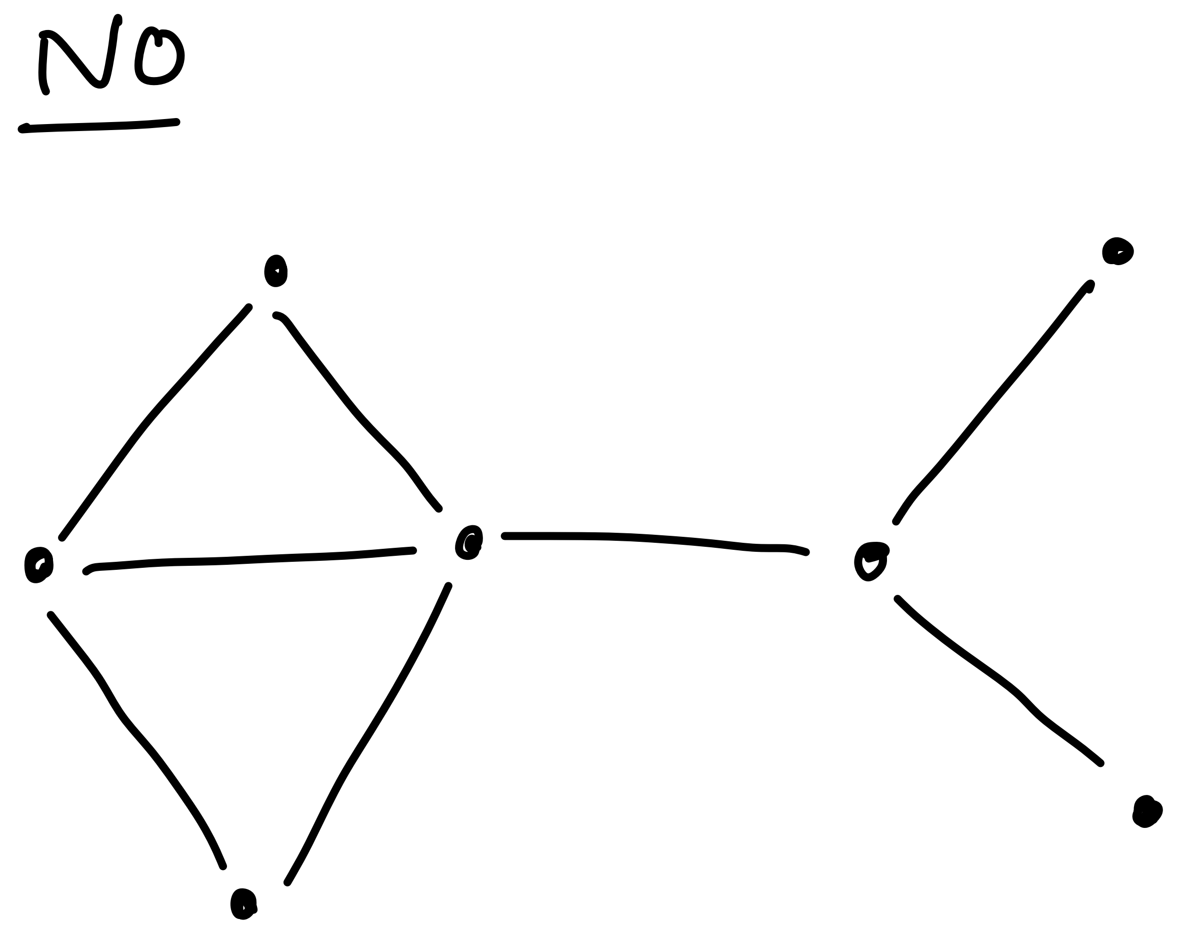
- Input: 3-CNF formula φ. Output: If there exists a $z \in \{0,1\}^n$ such that $\varphi(z) = 1$.
- Proof: the satisfying assignment x
- **Certifier algorithm** $\mathscr{V}\left(\varphi, z\right)$:
    - Check that every disjunction (OR statement) is true.

Examples of problems in NP

Hamiltonian Path

- Input: G. Output: Does there exists a simple path that visits every vertex (i.e. without repeating vertices)?
- Proof: The permutation π listing the sequence of vertices in the path.
- **Certifier algorithm** $\mathscr{V}(G, \pi)$:
    - Check that every entry of π is distinct.
    - Check that each (π_i, π_{i+1}) is an edge of E.

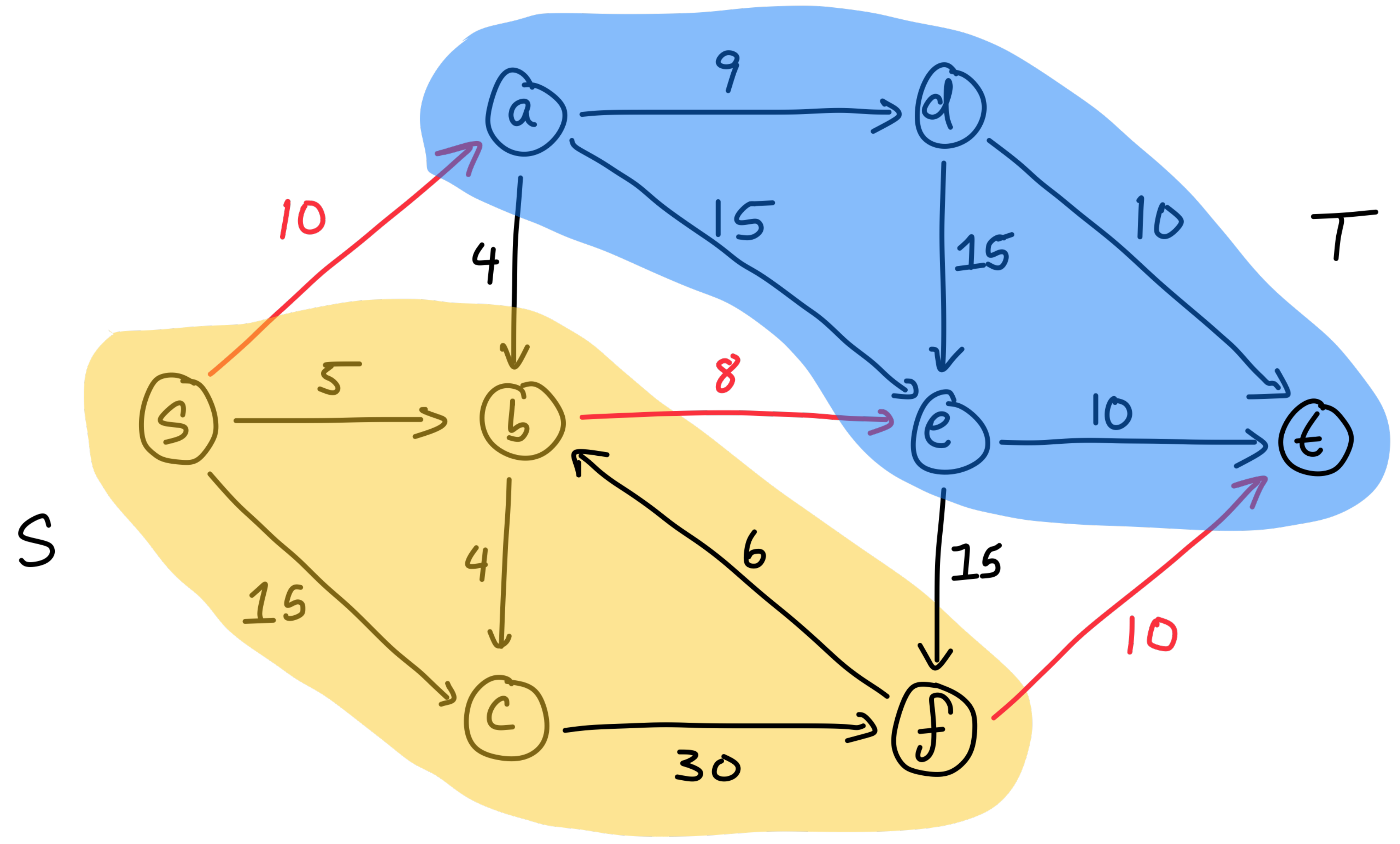




Examples of problems in NP

Min cut

- Input: (G, c, s, t, k). Output: Is there a min cut of size $\leq k$?
- Proof: $\pi \in \{0,1\}^V$ describes the vertices in S for an s-t cut (S, T)
- **Certifier algorithm** $\mathscr{V}(x, \pi)$:
  - Check that $\pi_s = 1$ and $\pi_t = 0$ (valid s-t cut).
  - Compute, $c(S, T) = \sum_{(u,v)\in E\ :\ \pi_u=1,\pi_v=0} c(u, v)$
  - Check if $c(S, T) \leq k$
- **Therefore**, MINCUT $\in$ NP.



Examples of problems in NP
Min cut

- Input: (G, c, s, t, k). Output: Is there a min cut of size $\leq k$?
- Proof: empty string
- **Certifier algorithm** $\mathscr{V}(x)$:
    - Compute optimal s-t cut (S, T) using Edmonds-Karp flow algorithm.
    - Compute, $c(S, T) = \sum_{(u,v)\in E\ :\ \pi_u=1, \pi_v=0} c(u, v)$
    - Check if $c(S, T) \leq k$
- **Therefore,** MINCUT $\in$ NP.

Examples of problems in NP

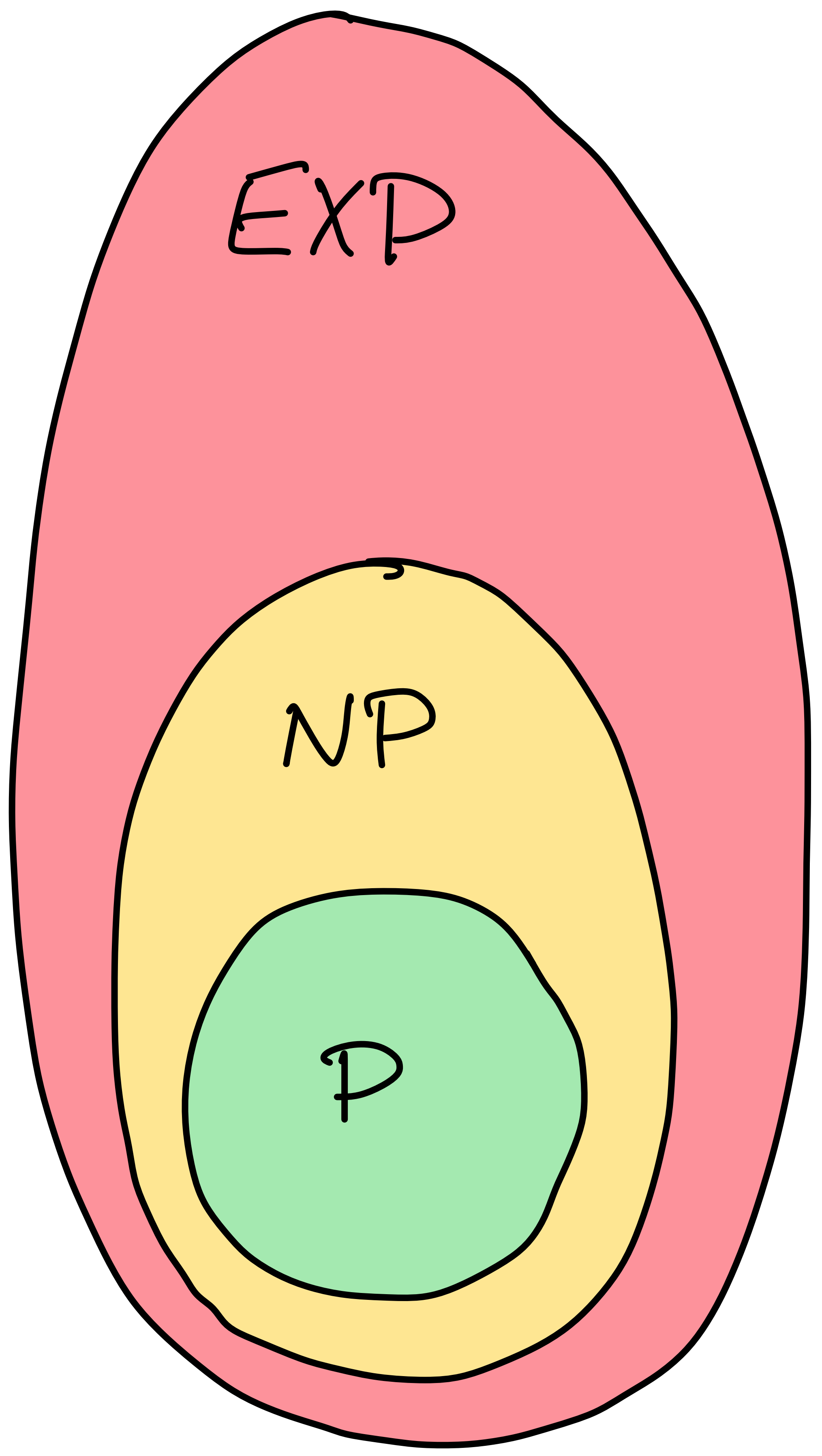
Min cut

- Input: (G, c, s, t, k). Output: Is there a min cut of size $\leq k$?
- Proof: empty string
- **Certifier algorithm** $\mathscr{V}(x)$:
  - Compute optimal s-t cut (S, T) using Edmonds-Karp flow algorithm.
  - Compute, $c(S, T) = \sum_{(u,v) \in E \,:\, \pi_u = 1, \pi_v = 0} c(u, v)$
  - Check if $c(S, T) \leq k$
- **Therefore**, MINCUT $\in$ P.

P, NP, **and,** EXP

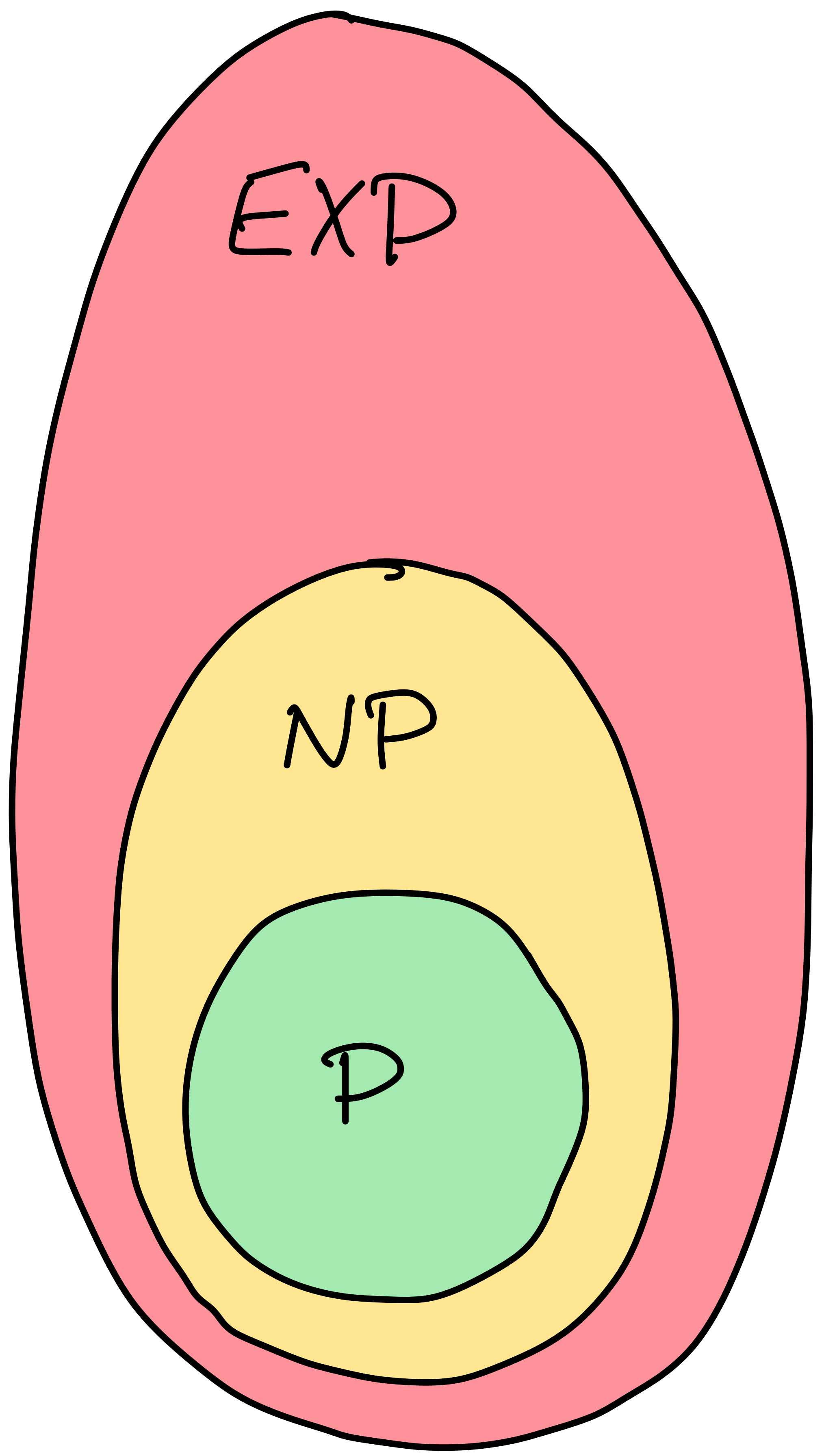
- P : decision problems with a poly-time algorithm
- NP : decision problems with a poly-time certifier
- EXP : decision problems with a exp-time algorithm

- **Theorem:** P $\subseteq$ NP
- **Theorem:** NP $\subseteq$ EXP



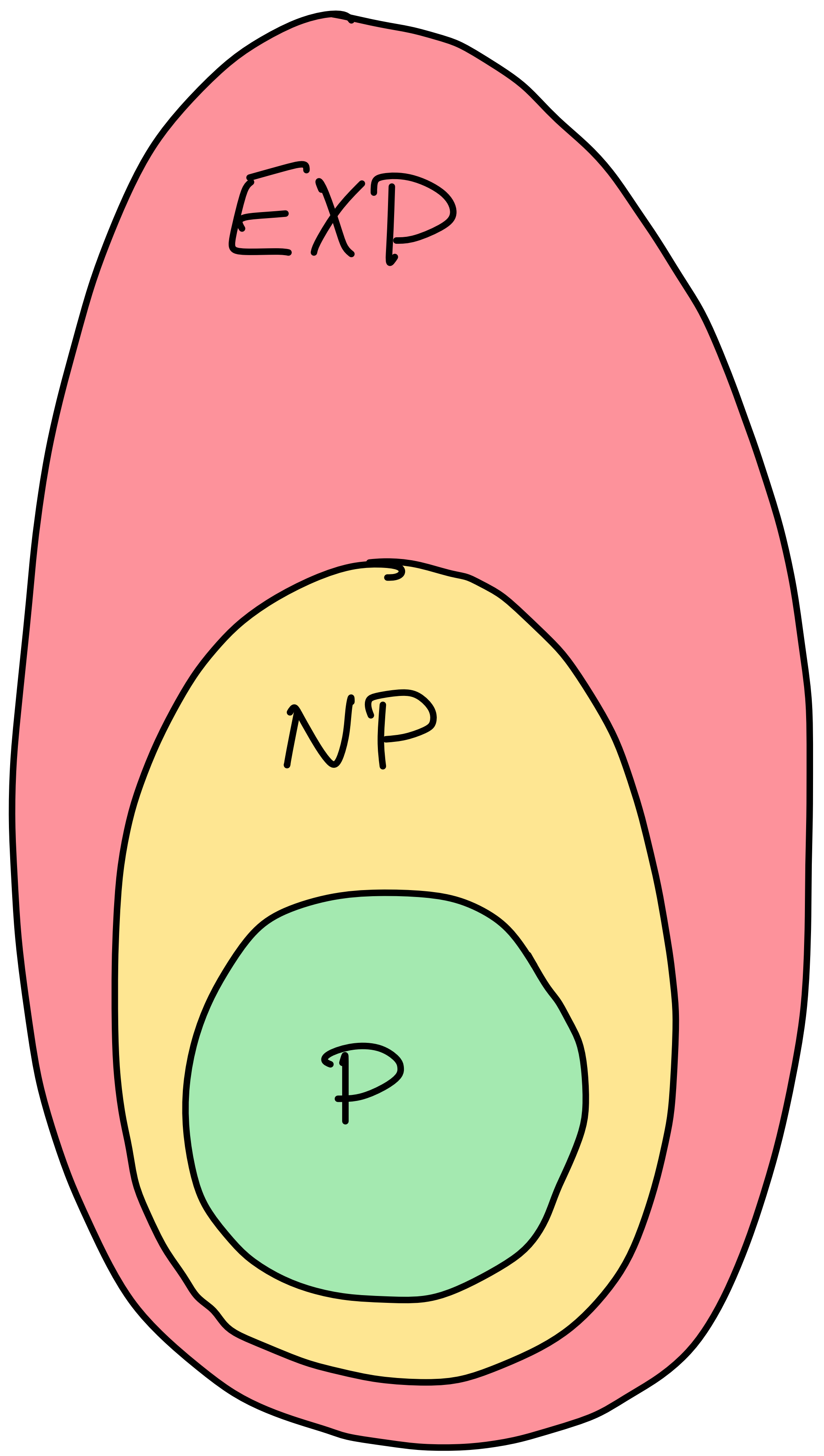
P ⊆ NP

- **Proof:**
  - Consider any problem X in P.
  - By definition, there exists an algorithm $\mathscr{A}(x)$ solves X.
  - Proof: $\pi =$ empty string. Certifier $\mathscr{V}(x, \pi) = \mathscr{A}(x)$.



NP $\subseteq$ EXP

- **Proof idea:** Brute-force search over all possible proofs π.
- **Proof:**
  - Consider any problem X in NP.
  - By definition, there exists a certifier $\mathscr{C}(x, \pi)$ for X such that $|\pi| \leq |x|^{c'}$.
  - To solve the problem on input x:
    - For all $\pi \in \{0,1\}^{|x|^{c'}}$, run $\mathscr{C}(x, \pi)$ and return "yes" if $\mathscr{C}(x, \pi) = 1$.
    - Otherwise, return "no".
- This exhaustively iterates over all possible certificates.



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

- Is the decision problem of solving every problem is as easy as the certification problem?
- There is a $1 million bounty for solving the problem (in either direction!)

- If yes: There is an efficient/poly-time algorithm for <u>every</u> NP problem
- If no: No efficient/poly-time algorithm for some problems such as 3-COLOR, TSP, 3-SAT, KNAPSACK, VERTEX-COVER, SET-COVER, HAM-CYCLE, …

How do we decide if a problem is in NP?

- What might a solution look like for the problem?
- Given a solution, is it easy to check that the solution is correct? Is it easy to detect a mistake in the solution?
- If the solution is polynomial-length in the input of the problem then the problem is in NP
- Most proofs of containment in NP are ridiculously simple
    - See the examples from section for how short our proofs are
    - Usually the hard problem is proving that problems are NP-complete — i.e., the hardest NP problems

What is the “hardest” problem in NP?

- First of all, what does it mean for a problem to be hard?
- **Intuition:** Let us say that a problem B is **as hard as** than a problem A if a fast algorithm for a problem B implies a fast algorithm for a problem B.
    - Example: Max flow is as hard as bipartite matching.
    - Example: Breadth-first search is as hard as 2-coloring graphs.
    - Example: Constructing generators and power grids is as hard as MST.
    - Example: Max flow is exactly as hard as min cut.
- If an instance of problem B can be efficiently converted into an instance of a problem A this is called a **reduction**.

Reductions throughout this class

- We’ve seen reductions many times before in this class
- Anytime you used an algorithm as a subroutine — you were *morally* performing a reduction
- Examples:
  - Bipartite matching as a flow problem
  - Ship port assignment as a stable matching
  - Little Johnny walking to his mother’s house as a shortest path problem

Example of a reduction

Subset Sum $\leq_p$ Decision-Knapsack

- Subset Sum: Give input $a_1, \ldots, a_n, T$, decide if there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} a_i = T$.
- Decision-Knapsack: Given input $w_1, \ldots, w_n, v_1, \ldots, v_n, W, V$, 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 $\mathscr{A}'$ for solving Subset Sum from an algorithm $\mathscr{A}$ for solving Knapsack.
  - Given input $a_1, \ldots, a_n, T$, define $w_i = v_i \leftarrow a_i$ and $W \leftarrow T, \quad V \leftarrow T$.
  - Then run $\mathscr{A}$ on $(w_1, \ldots, w_n, v_1, \ldots, v_n, W, V)$.

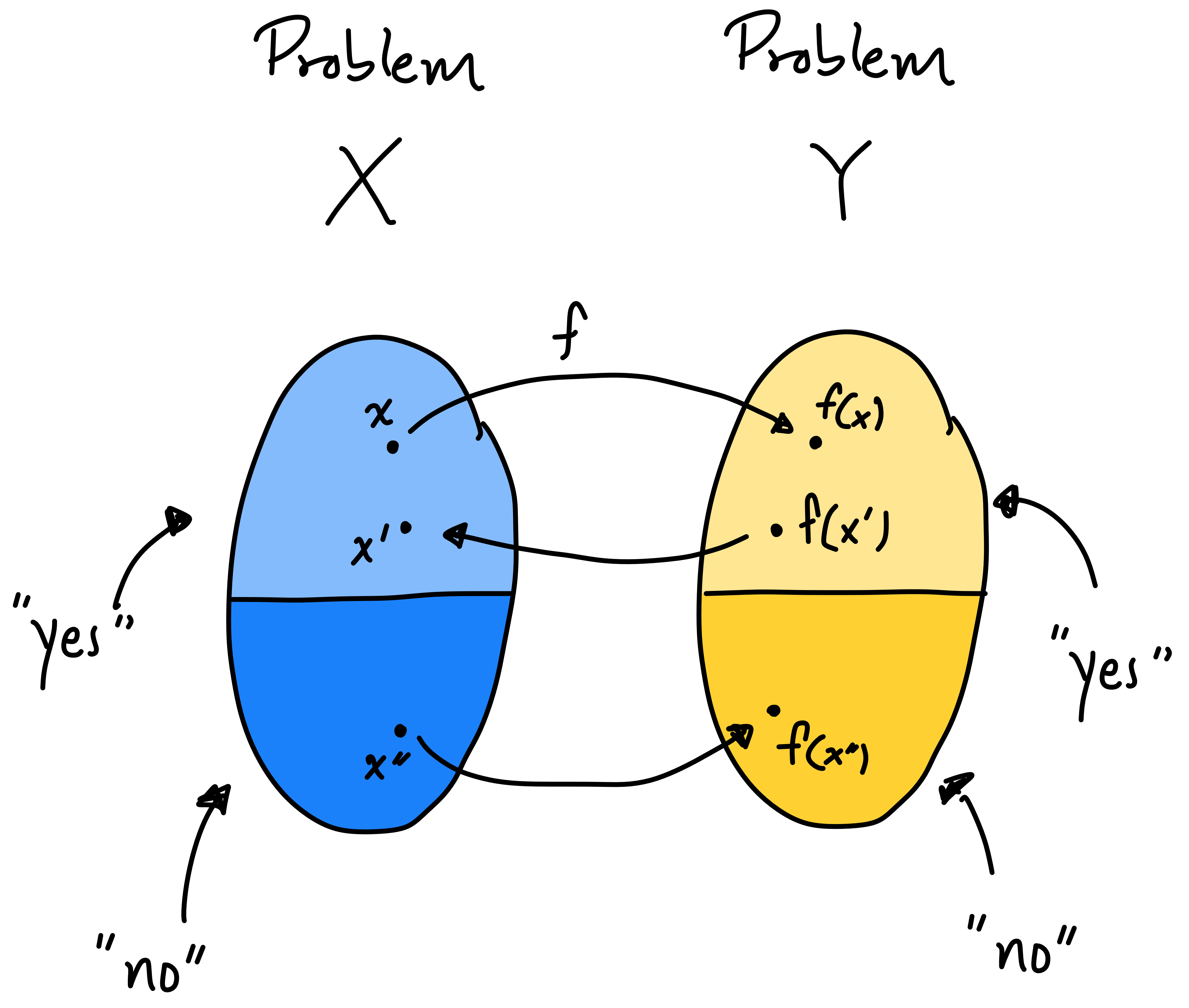
Example of a reduction

Subset Sum $\leq_p$ Decision-Knapsack

- $x = (\vec{a}, T) \in$ Subset Sum, $f(x) = (\vec{w} = \vec{v} \leftarrow \vec{a}, W = V \leftarrow T)$
- If x is a "yes" instance, then there exists $S \subseteq [n]$ s.t. $\sum_{i \in S} a_i = T$
  - Therefore, $\begin{array}{l} \sum_{i \in S} w_i = \sum_{i \in S} a_i = T \leq W, \\ \sum_{i \in S} v_i = \sum_{i \in S} a_i = T \geq V. \end{array}$ So $f(x)$ is a "yes" instance.
- If $f(x)$ is a "yes" instance, then there exists $S \subseteq [n]$ s.t. $\sum_{i \in S} w_i \geq W$ and $\sum_{i \in S} v_i \leq V$.
  - So $T = V \leq \sum_{i \in S} v_i = \sum_{i \in S} a_i \leq \sum_{i \in S} w_i \leq W = T$, proving that x is a "yes" instance.

Proving a reduction is correct

- The previous example is a reduction between Subset Sum and D-Knapsack
- To generate a reduction $X \leq_p Y$ between two decision problems X and Y
  - We need to find a **poly-time computable** function $f : X \to Y$ that converts instances x of X into instances $f(x)$ of Y
    - Morally, f is the precompute we would apply before the algorithm for Y
  - If for every $x \in X$ that is a "yes", then $f(x) \in Y$ is also a "yes" instance
  - If for every $f(x') \in Y$ that is a "yes", then $x' \in X$ is also a "yes" instance
    - Equiv. to: If $x'' \in X$ is a "no", then $f(x'') \in Y$ is a "no"



NP-**completeness**

- **Simple definition:** A problem is NP-complete problem if (a) it is in NP and (b) it is the "hardest" problem in NP
- **Necessary consequence (we will show soon):** A problem X is NP-complete iff
  - If X has a poly-time algorithm, then every problem in NP has a poly-time algorithm.
  - If some problem $Y \in$ NP does not have a poly-time algorithm, then neither does X.
- **Punchline:** Once we show Knapsack is NP-complete, then if you find a way to solve Knapsack in poly-time, then you will have solved every problem in NP in poly-time.

NP-completeness

- Proving that a problem Y is the hardest problem in NP requires showing
  - that if there exists a poly-time algorithm $\mathscr{A}$ for solving Y, then for any problem $X \in$ NP, there exists a poly-time algorithm $\mathscr{A}'$ for solving X
  - This is the **reduction** of X to Y. We denote this by $X \leq_p Y$.
- Formally, we say X reduces to Y (denoted $X \leq_p Y$) if any instance x of X can be solved by the following algorithm:
  - In $\text{poly}(|x|)$ time, compute $y = f(x)$, an instance of the problem Y
  - Run a subroutine to decide if y is a "yes" instance of Y — returning the answer exactly
  - This is known as a Karp or many-to-one reduction.

NP-**completeness**

- **Formal definition:** A problem Y is NP-complete if $Y \in$ NP and for every problem $X \in$ NP, $X \leq_p Y$.
- **Theorem:** Let Y be a NP-complete problem. Then Y is solvable in poly-time iff P $=$ NP.
- **Proof:**
  - ($\Longleftarrow$) If P $=$ NP, then Y has a poly-time algorithm since $Y \in$ NP.
  - ($\Longrightarrow$) 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 $X \in$ P. So P $=$ NP.
- **Fundamental question:** Do there exist “natural” NP-complete problems?

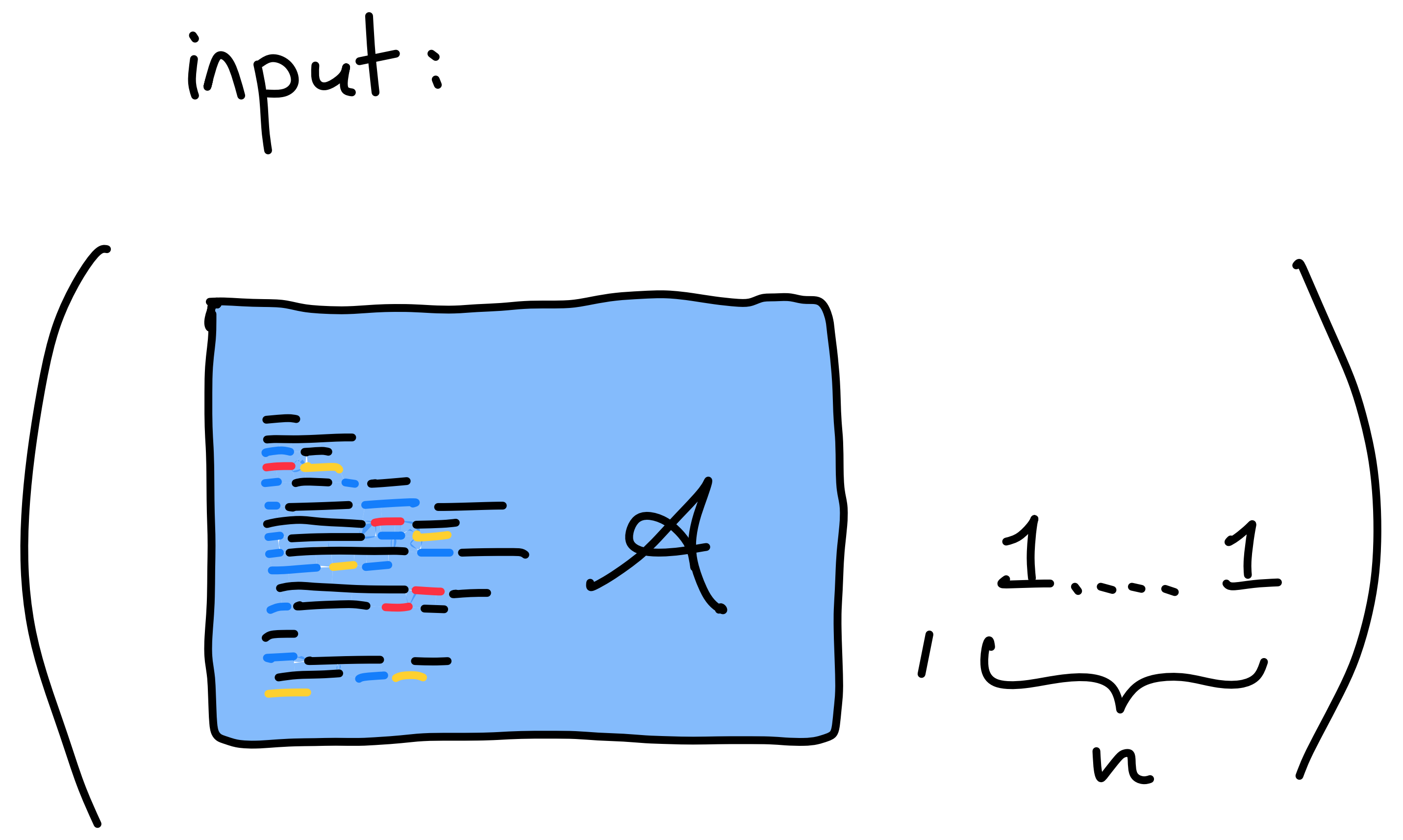
A list of NP-complete problems

- Boolean function satisfiability
- 0-1 Integer programming
- Graph problems: Vertex cover, 3-color, independent set, set cover, max cut
- Path and cycle problems: Hamiltonian path, traveling salesman
- Combinatorial optimization problems: Knapsack, Subset sum

The “first” NP-complete problem

Satisfiability

- **Satisfiability:**
  Input: $(\langle \mathscr{A} \rangle, n)$, the description of a decision algorithm $\mathscr{A}$ and integer n in unary.
  Output: Whether there exists a π such that $\mathscr{A}(\pi) = 1$ and $|\pi| = n$.
- **Theorem:** Satisfiability is NP-complete.
- **Proof:**
  - Satisfiability is in NP as π is a proof of the satisfiability.
  - For any other problem $X \in$ NP, there exists a certifier $\mathscr{V}(x, \pi)$ such that x is a “yes” instance iff there exists a π such that $\mathscr{V}(x, \pi)$ accepts.
    - Let $n = |\pi|$ taken as input by $\mathscr{V}$.
    - Define $\mathscr{A}(\pi) :=$ as the poly-sized program computing $\mathscr{V}(x, \pi)$.
    - Then x is a “yes” instance iff exists a π such that $\mathscr{A}(\pi) = 1$ and $|\pi| = n$.
    - So $X \leq_p Y$, proving NP-completeness.



Proving more NP-complete problems

- **Recipe** for showing that problem Y is NP-complete
  - Step 1: Show that $Y \in$ NP.
  - Step 2: Choose a known NP-complete problem X.
  - Step 3: Prove that $X \leq_p Y$.
- **Correctness** of recipe: We claim that $\leq_p$ is a transitive operation.
  - If $W \leq_p X$ and $X \leq_p Y$ then $W \leq_p Y$.
  - For any problem $W \in$ NP, then $W \leq_p Y$, proving that Y is NP-complete.