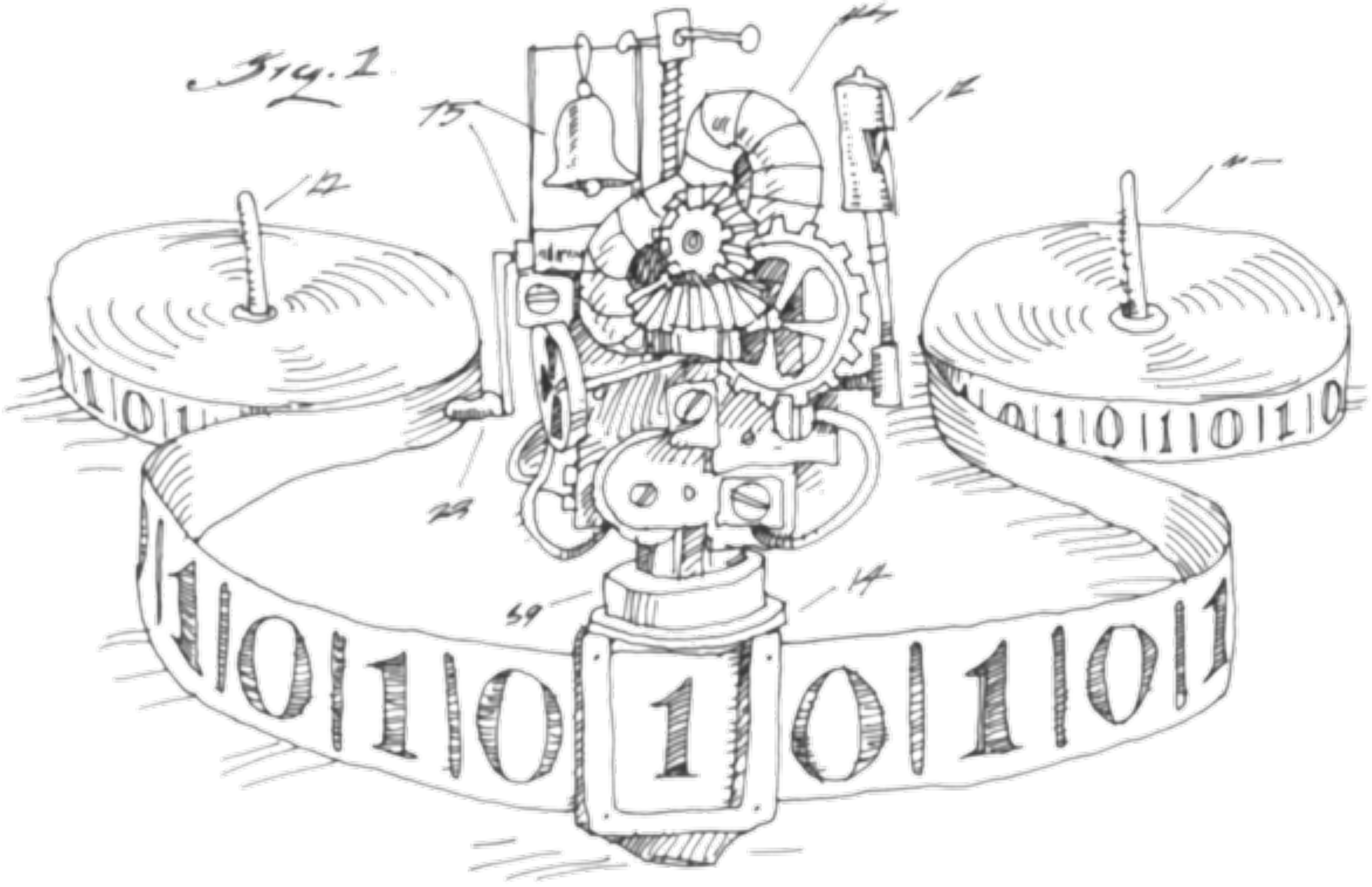
计算复杂性
computational complexity

钮鑫涛
Nanjing University
2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

Model for Computation—Turing Machine

- An infinite **tape** divided into cells.
- A head that can **read** or **write** symbols on the tape, and **move** the tape left or right one cell at a time.
- A state register storing current **state** of the machine, among finitely many states.
- A *finite* **table** of **instructions**:
  - Given current state and current read symbol:
    - Either erase or write a symbol;
    - Move the head (left, right, or remain stationary);
    - Stay the same state or change to a new state.



Decision Problem

- Decision problem: problems that expect a **YES** or **NO** answer.
  - An **instance** of decision problem conceptually contains two parts:
    - Instance description;
    - The question itself.
  - For such problems, we can split all possible instances into two categories: **YES-instances** (whose correct answer is YES) and **No-instances** (whose correct answer is NO).
- **Example:**
  - Given a graph G, a pair of nodes (u, v), an integer k, is every path between (u, v) of length at least k?
  - Given a multiset S, is there a way to partition S into two subsets of equal sum?

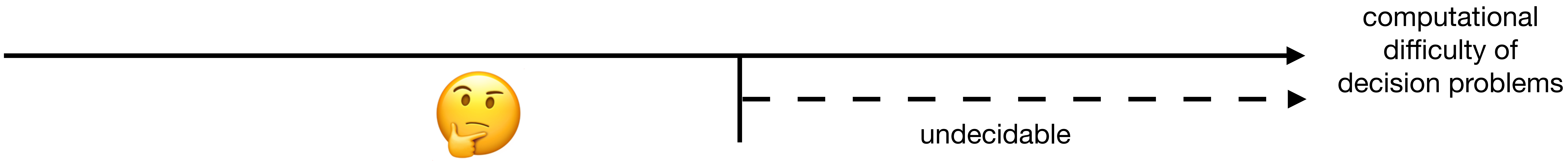
Optimization vs Decision

- In an **optimization problem**, among all feasible solutions, we find one that *maximizes* (or *minimizes*) a given *objective*.
  - Example: Given a graph G, a pair of nodes (u, v), what is the length of the shortest path between (u, v)?
- If we have an efficient algorithm for a decision problem, then we can usually solve the corresponding optimization problem efficiently, and vice versa.
  - Example: Given a graph G, a pair of nodes (u, v), an integer k, is every path between (u, v) of length at least k?
  - Another example: chromatic number vs k-colorable.

Computability

- For **each decision** problem, there exists a TM to decide it?
  - Informally, we say a TM solves (decides) a decision problem if for *each* instance of the problem, within *finite* steps, the TM *correctly* outputs “yes” or “no” and then *halts*.
  - No! E.g., The halting problem.

Problems can be solved in practice

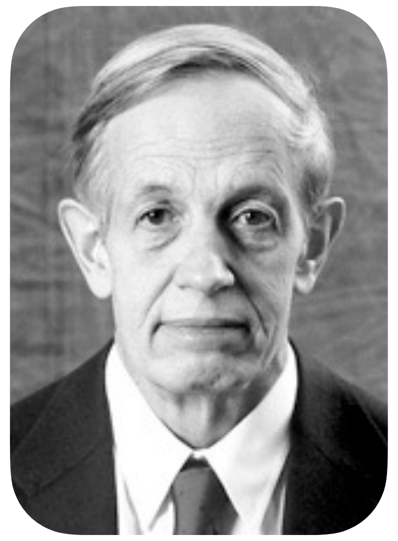


- For these **computable** problems, can all of them be solved efficiently in practice?
- For a given decision problem, can TM decide it quickly?

Problems can be solved in practice

- Which problems will we be able to solve in practice?
  - A working definition. Those with poly-time algorithms.

von Neumann
(1953)



Nash
(1955)

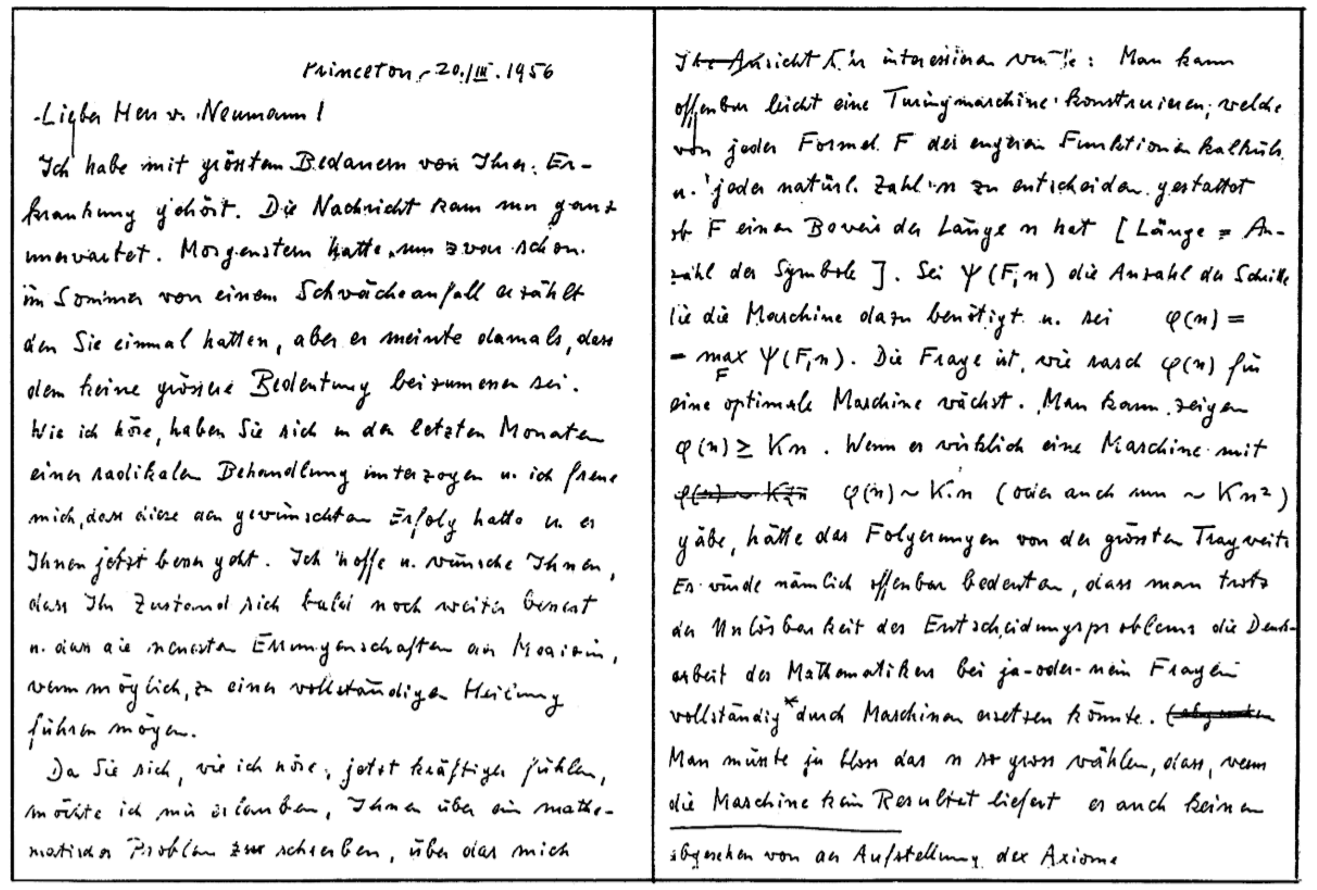
Gödel
(1956)

Cobham
(1964)

Edmonds
(1965)

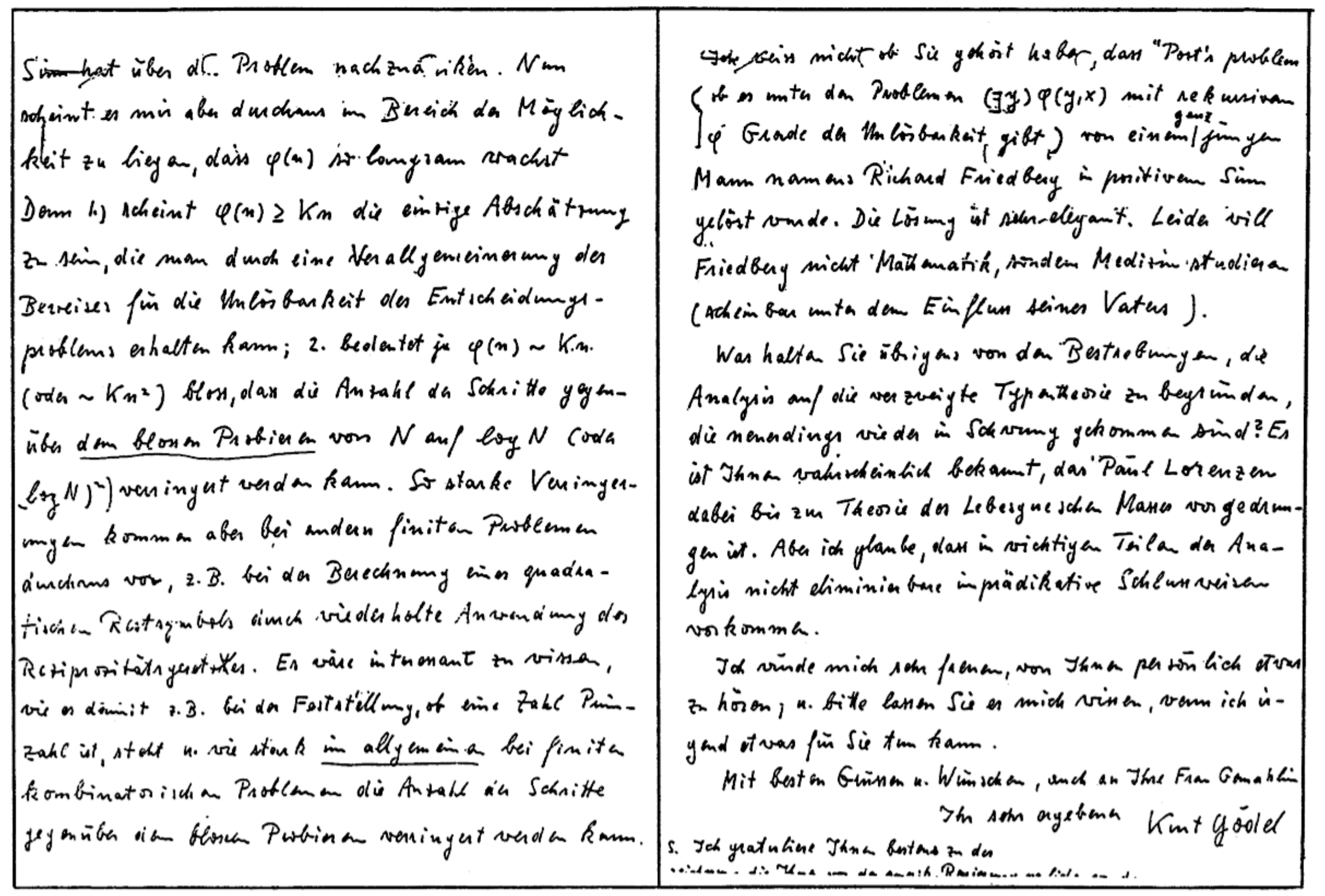
Rabin
(1966)

In a 1956 letter, Göedel asked von Neumann about the computational complexity of an NP complete problem



Princeton, 20./III. 1956

Lieber Herr v. Neumann!



Kurt Gödel

The Class P

- Consider a decision problem $\mathscr{P}$, let I be an instance of $\mathscr{P}$.
- Let $|I|$ denote the length of I under, say, binary encoding.
- An algorithm $\mathscr{A}$ for $\mathscr{P}$ is **polynomially bounded**, if the runtime of $\mathscr{A}$ is $(|I|)^{O(1)}$ for all I.
- **P** is the set of decision problems each of which has a polynomially bounded algorithm.
- **P** is the set of decision problems each of which can be decided by some TM within polynomial time.
- Most (but not all) problems we have studied so far are in **P**.

Some notes on P

- **P** contains the set of so-called tractable problems.
- So problems with $\Theta(n^{100})$ time algorithms also tractable?
  - Being in **P** doesn't mean a problem has efficient algorithms.
- Nonetheless:
  - Problems not in **P** are definitely expensive to solve.
  - Problems in **P** have "closure properties" for algorithm composition.
  - The property of being in **P** is independent of computation models.

A note on size of input

- Recall decision problem $\mathscr{P} \in$ **P** if there exists an algorithm that can solve $\mathscr{P}$ in $(|I|)^{O(1)}$ time for every instance I of $\mathscr{P}$.

IsPrime(n):

for $i := 2$ **to** n - 1
    if $n\%i = 0$
        return $False$
return $True$

Normally we assume the encoding:
- of an integer is polynomially related to its binary representation
- of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas.

- This algorithm has poly-n runtime, so **Primes** $\in$ **P** ?

- No! The size of the input is $O(\log n)$ with binary encoding.

- Indeed **Primes** $\in$ **P**, but proved with a different algorithm
(AKS primality test by Agrawal, Kayal, and Saxena)

Subset Sum

- **Problem:** Given an array $X[1\cdots n]$ of n **positive** integers, can we find a subset in X that sums to given integer T?
- **Step 1**: Characterize the structure of solution.
  - If there is a solution S, either $X[1]$ is in it or not.
- **Step 2**: Recursively define the value of an optimal solution.
  - Let $ss(i,t)$ = true iff instance "$X[i...n], t$" has a solution.
  - $$ss(i,t) = \begin{cases} true & \text{if } t = 0 \\ ss(i+1,t) & \text{if } t < X[i] \\ false & \text{if } i > n \\ ss(i+1,t) \vee ss(i+1,t-X[i]) & \text{otherwise} \end{cases}$$
- **Step 3**: Compute the value of an optimal solution (Bottom-Up).
  - Build an 2D array $ss[1...n,0...T]$
  - Evaluation order: bottom row to top row; left to right within each row.

Runtime is $O(nT)$

SubsetSumDP(X,T):

```
ss[n, 0] := True
for t := 1 to T
    ss[n, t] := (X[n] = t) ? True : False
for i := n - 1 downto 1
    ss[i, 0] := True
    for t :=1 to X[i] - 1
        ss[i, t] := ss[i + 1, t]
    for t := X[i] to T
        ss[i, t] := Or(ss[i + 1, t], ss[i + 1, t - X[i]])
return ss[1,T]
```

Subset Sum

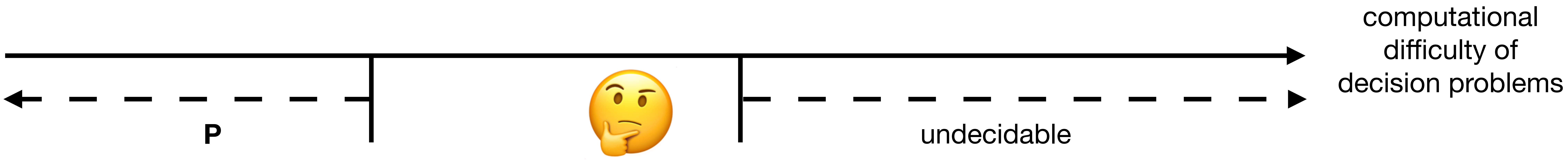
- **Problem:** Given an array $X[1 \cdots n]$ of n **positive** integers, can we find a subset in X that sums to given integer T?

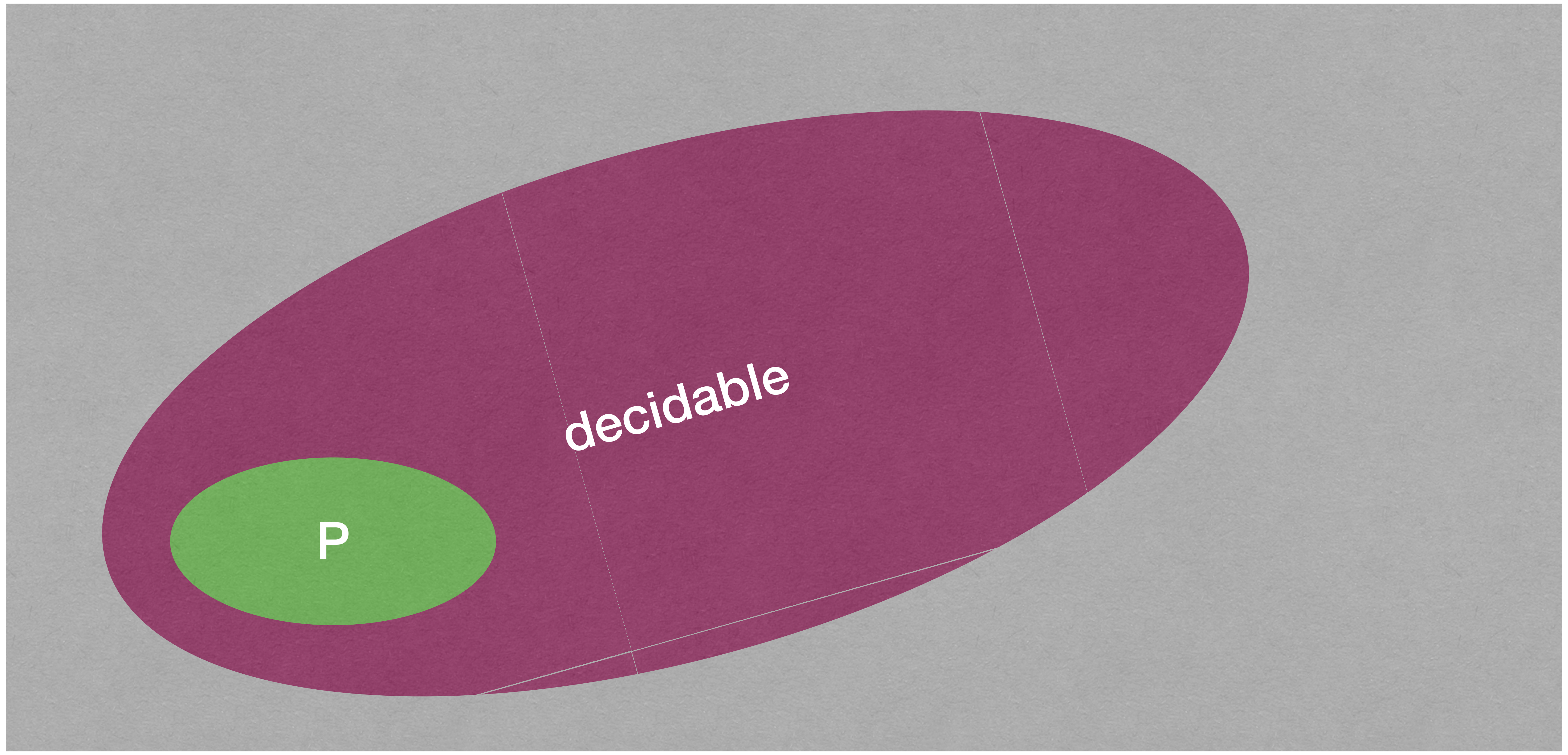
- Simple solution: recursively enumerates all 2^n subsets, leading to an algorithm costing $O(2^n)$ time.

- Dynamic programming: costing $O(nT)$ time.

- Both algorithms are not polynomial time algorithms!

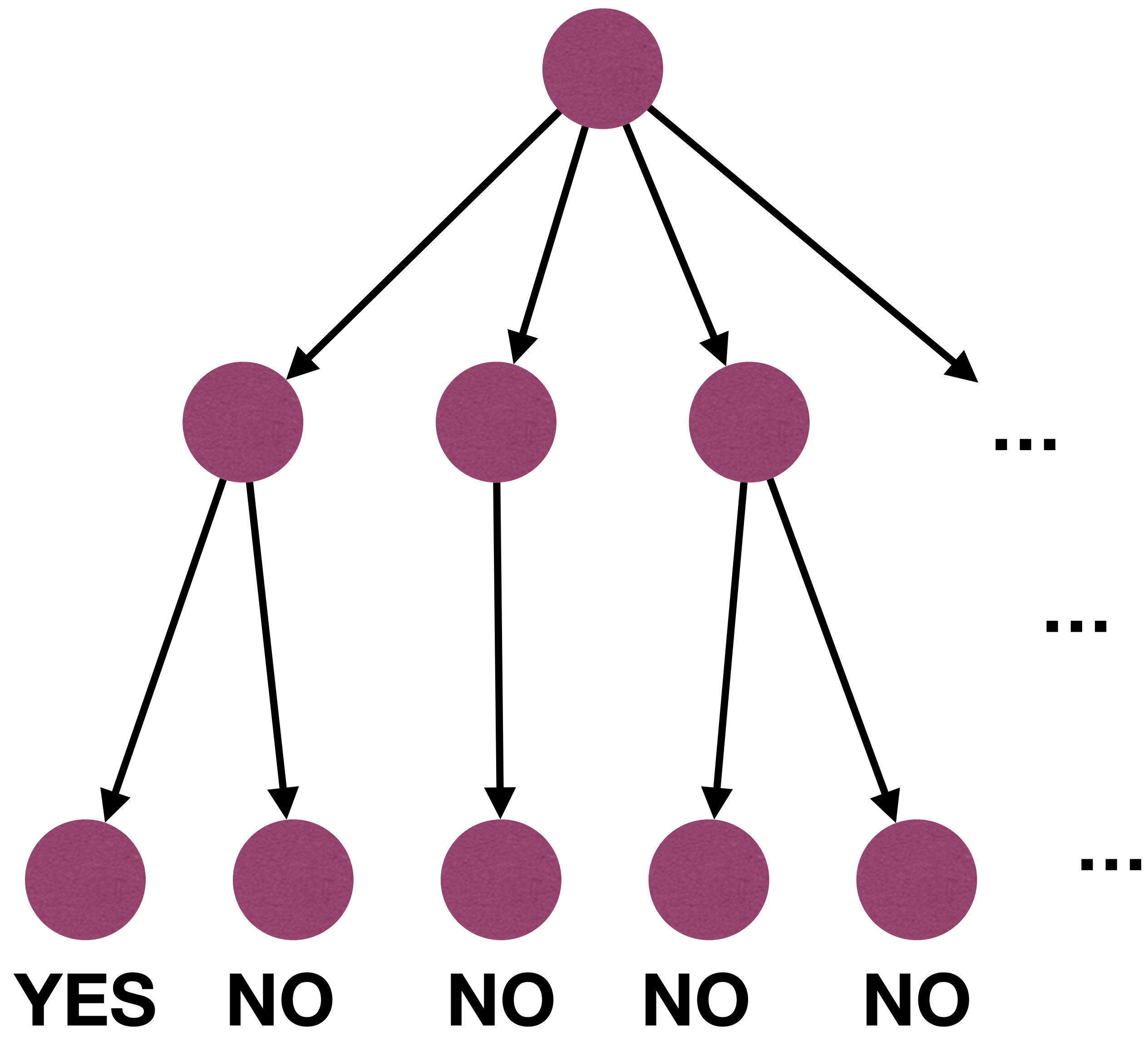
Problems can be solved in practice





Non-deterministic Turing Machine

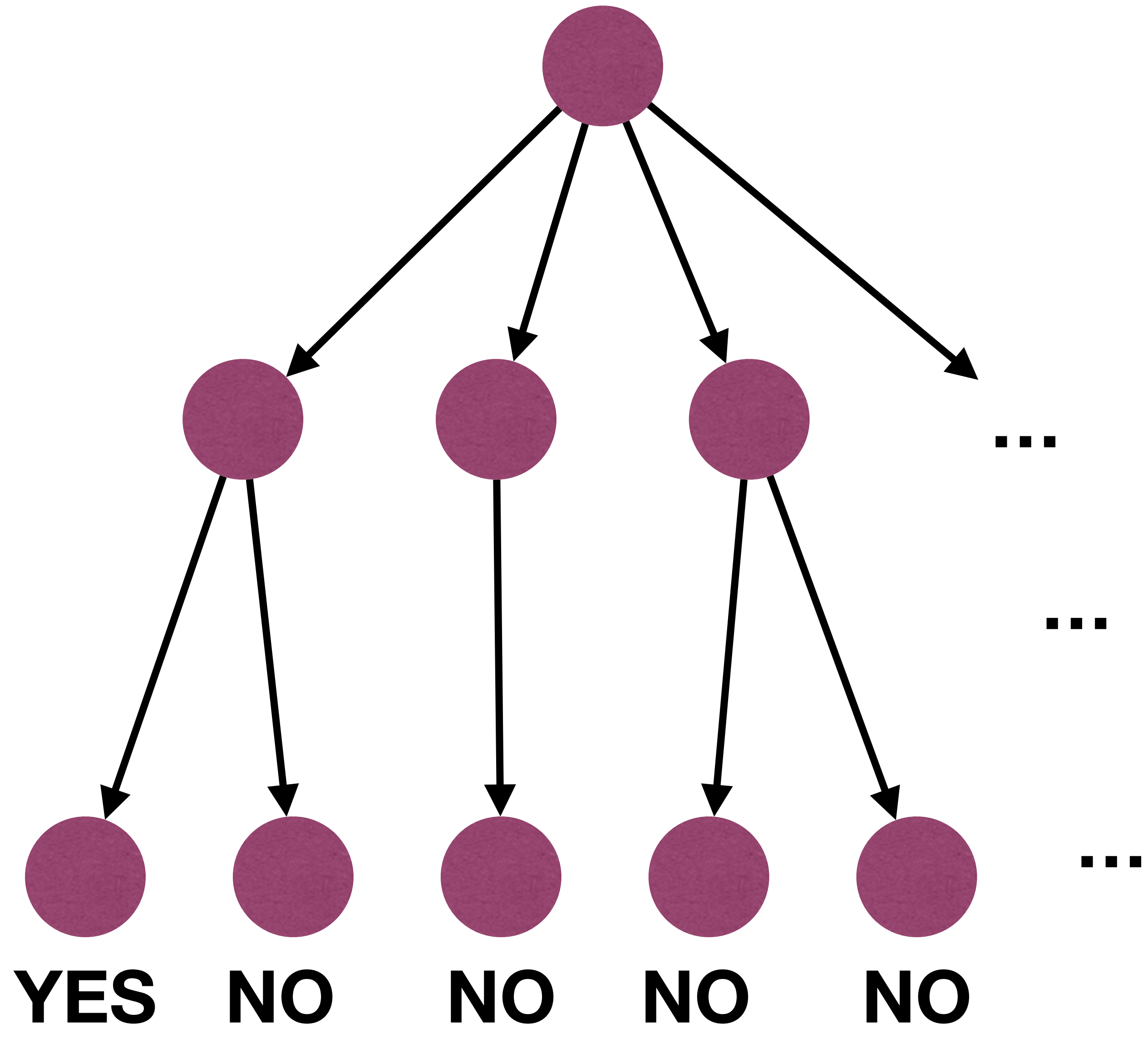
- An infinite tape divided into cells.
- A head that can read or write symbols on the tape, and move the tape left or right one cell at a time.
- A state register storing current state of the machine, among finitely many states.
- A *finite* **table** of instructions:
    - Given current state and current read symbol, there are ***many*** actions can be chosen — Nondeterminism!
    - Nondeterminism can be viewed as a kind of parallel computation wherein multiple independent processes or threads can be running concurrently.



The Class NP

- An Non-deterministic Turing Machine (NTM) M on input x returns "yes" iff some execution of $M(x)$ halts in "yes" state.

- Informally, we say an NTM solves (decides) a decision problem $\mathscr{P}$ in time $f(n)$ if for each instance I of $\mathscr{P}$ with $|I| = n$, within $f(n)$ steps, the NTM correctly returns "yes" or "no".
    - i.e., the height of the computation tree for I is no longer than $f(n)$.



- **NP** is the set of decision problems each of which can be decides by some NTM within polynomial time.

- **NP** means "non-deterministic polynomial time."

The Class NP, Take Two

- Let algorithm $C(I, t)$ is a "certifier" or "verifier" for problem $\mathscr{P}$ if for every instance I, I is a YES-instance iff there exists a string t such that $C(I, t) =$ `yes`.
  - Such string t is called a "certificate" or "witness" or "proof"
- Set of decision problems for which there exists a poly-time certifier.
  - If I is a YES-instance, then there exists t such that such that $C(I, t)$ **= yes**.
  - If I is a NO-instance, then for all t, $C(I, t)$ **= no**.
  - Note: the certificate t should have length polynomial in size of I.

The Class NP, Take Two

- Given a Boolean formula ϕ in CNF, is ϕ satisfiable?
  - Example: $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \overline{x_1}) \wedge (x_2 \vee \overline{x_1} \vee \overline{x_2}) \wedge (x_4)$
  - A certificate:
    - $x_1 =$ true, $x_2 =$ true, $x_3 =$ true, $x_4 =$ true
  - Certifier:
    - Sequentially evaluate each clause by assigning values (from the certificate) to each variable in that clause. If the values of all clauses are evaluated to be truth then return 1, otherwise return 0. (poly-time)

The Class NP, Take Two

- Theorem: **NP** equals the set of decision problems for which there exists a poly-time certifier.

- Proof:

  - $\Longrightarrow$ Suppose $p : \mathbb{N} \to \mathbb{N}$ is a polynomial and $\mathscr{P}$ is decided by a NTM N that runs in time $p(n)$. For every YES-instance I for $\mathscr{P}$, there must be a sequence of nondeterministic choices (i.e., a path in the computation tree) that makes N return **YES** on input I. We can use this sequence as a *certificate* for I. This certificate has length $p(|I|)$ and can be verified in polynomial time by a *deterministic* machine, which simulates the action of N using these nondeterministic choices and verifies that it would have been YES after using these nondeterministic choices. Thus, we have: **the set of decision problems for which there exists a poly-time certifier $\subseteq$ NP**

The Class NP, Take Two

- Theorem: **NP** equals the set of decision problems for which there exists a poly-time certifier.

- Proof:

  - $\Longleftarrow$ If for a decision problem $\mathscr{P}$ which has a poly-time certifier V, then we describe a polynomial-time NTM N that decides $\mathscr{P}$. On input I, it uses the ability to make nondeterministic choices to write down a string u of length $p(|I|)$ (the length of each path is at most $p(|I|)$, each path can be regarded as a candidate proof of I). Then it runs the deterministic verifier V to verify that u is a valid certificate for I, and if so, return true. Clearly, N returns true on I if and only if a valid certificate exists for I. Thus, we have:
  NP $\subseteq$ the set of decision problems for which there exists a poly-time certifier.

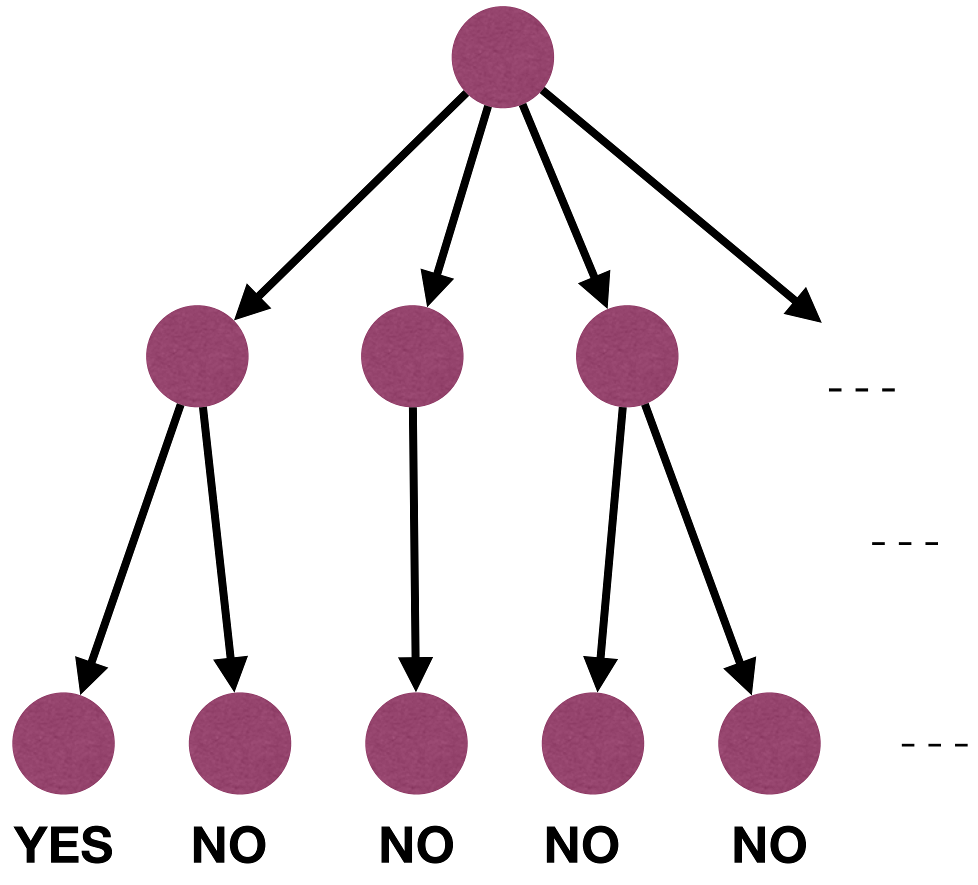
NP is the set of decision problems that
"yes" instances have short proofs that are efficiently verifiable

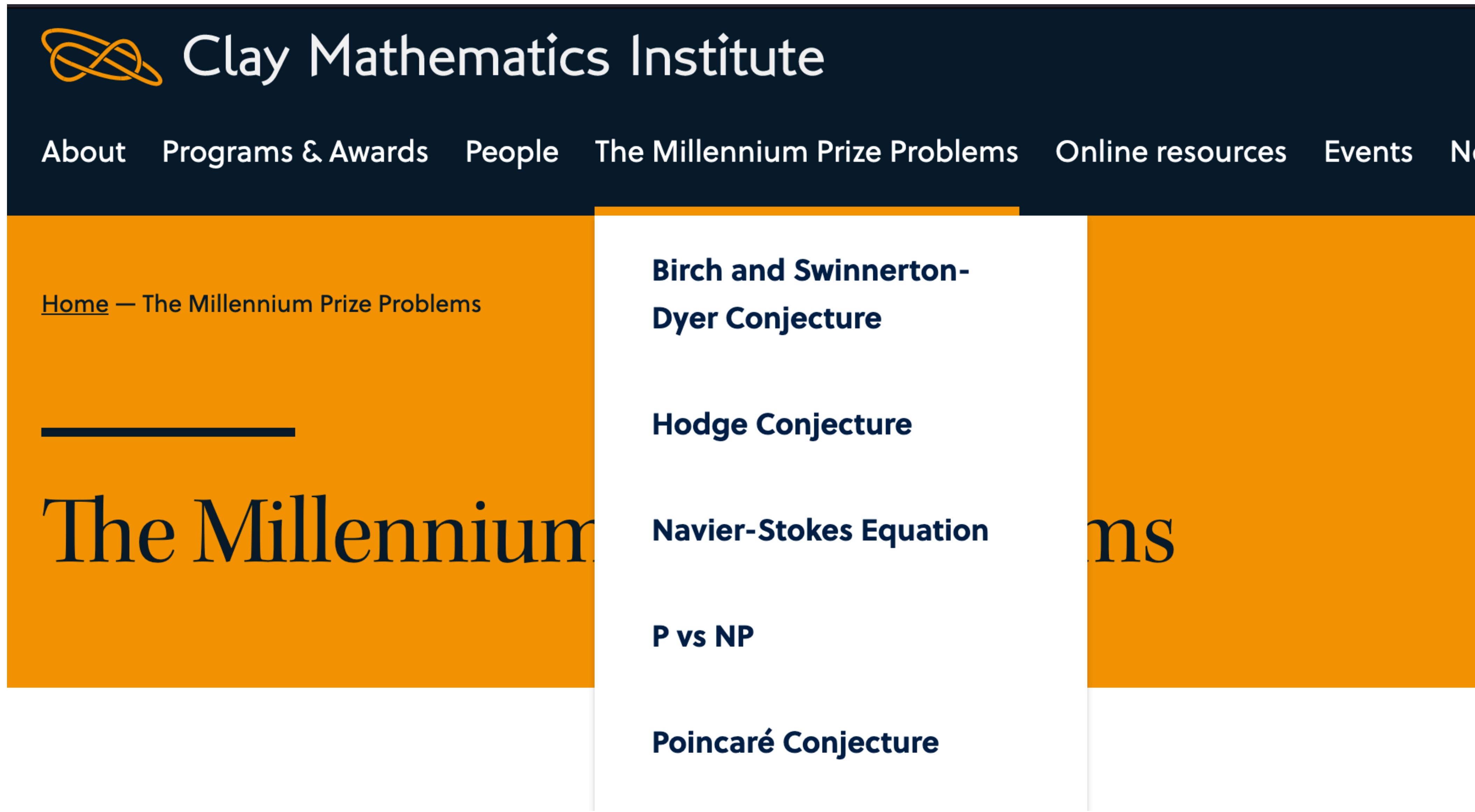
P $\subseteq$ NP

- **P** is the set of decision problems that have polynomially bounded algorithms.
- **P** is the set of decision problems that can be decided by (deterministic) TM within polynomial time.
- **NP** is the set of decision problems for which there exists a poly-time certifier.
- **NP** is the set of decision problems that can be decided by NTM within polynomial time.
- Any deterministic-algorithm is also a special non-deterministic algorithm, any TM is also a special NTM.

The big question $\mathbf{P} \neq \mathbf{NP}$

- Most people believe $\mathbf{P} \neq \mathbf{NP}$.
- Informally, NTM and non-deterministic algorithm allows exponential “trials” within polynomial time.



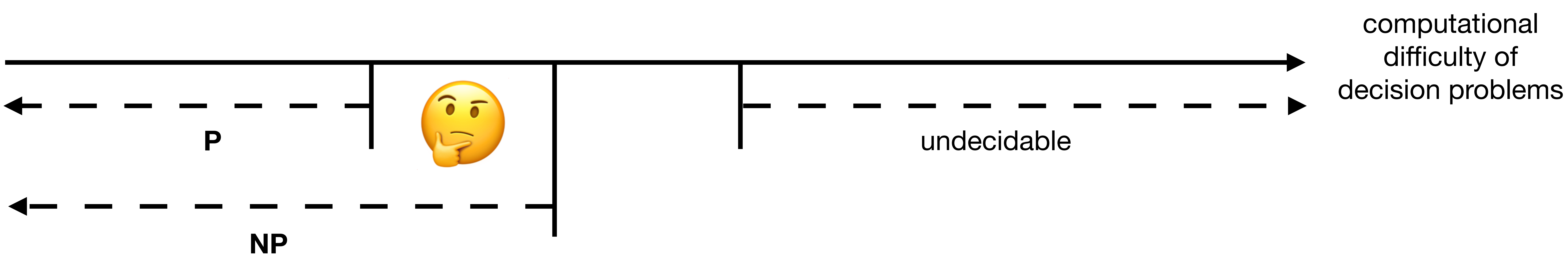


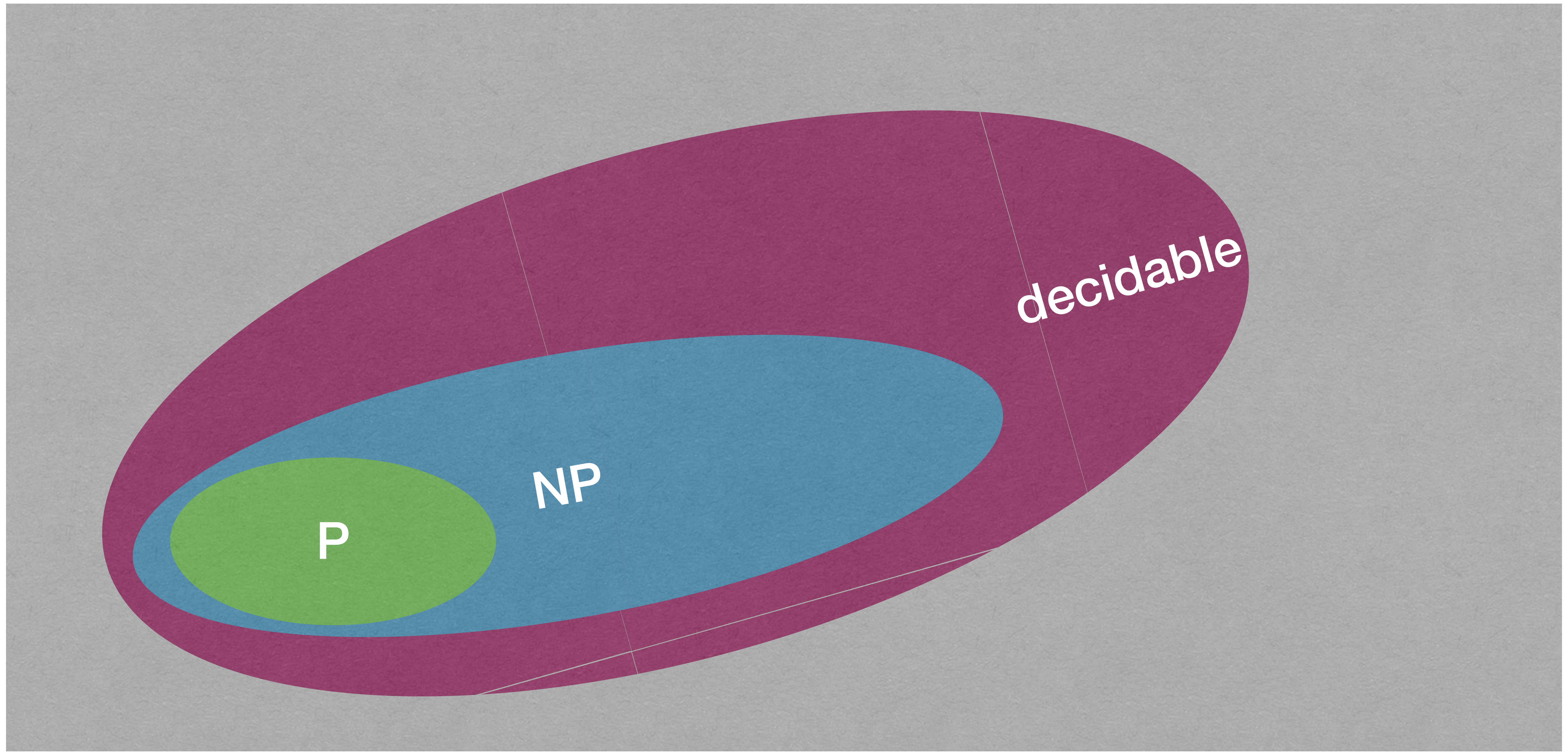
- P is the set of decision problems efficiently solvable.
- NP is the set of decision problems efficiently verifiable.

Solving a problem should be harder than verifying an answer?

Yet we haven't found any $\mathscr{P} \in \mathbf{NP}$, while $\mathscr{P} \notin \mathbf{P}$

If $P \neq NP$

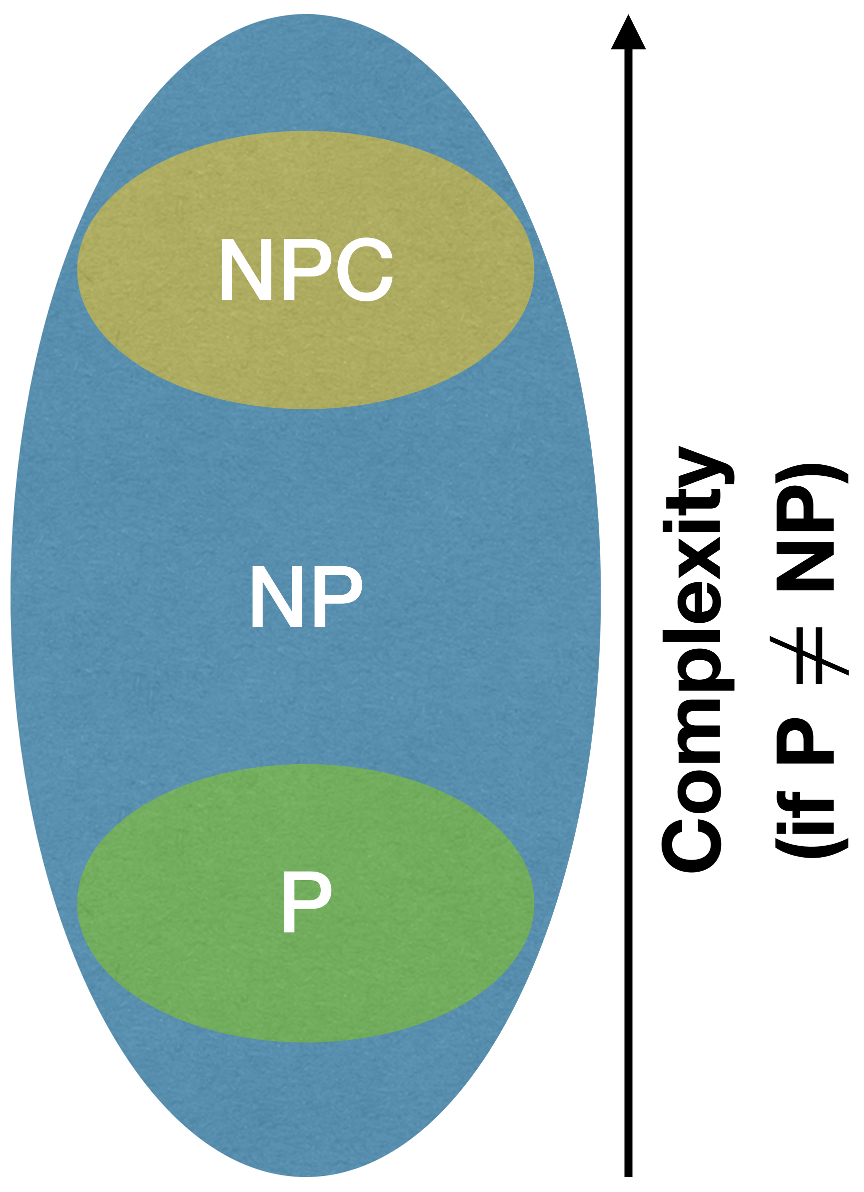

computational
difficulty of
decision problems
P
undecidable
NP


decidable
NP
P

NP completeness

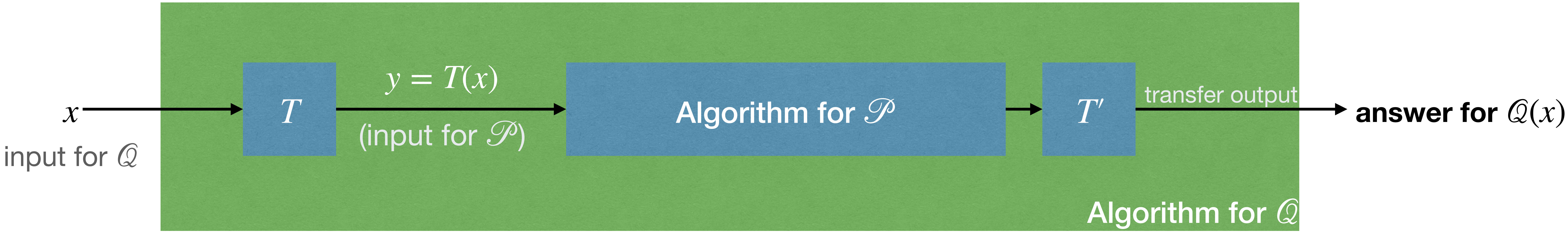
The hardest among the hard ones

- NP-Complete (**NPC**) problems are the **hardest** ones in NP.
- A decision problem $\mathscr{P}$ is **NPC** if:
  - The problem $\mathscr{P}$ is in **NP**.
  - If we have an algorithm for $\mathscr{P}$, then **all** problems in **NP** can be solved with limited extra work.



Reduction

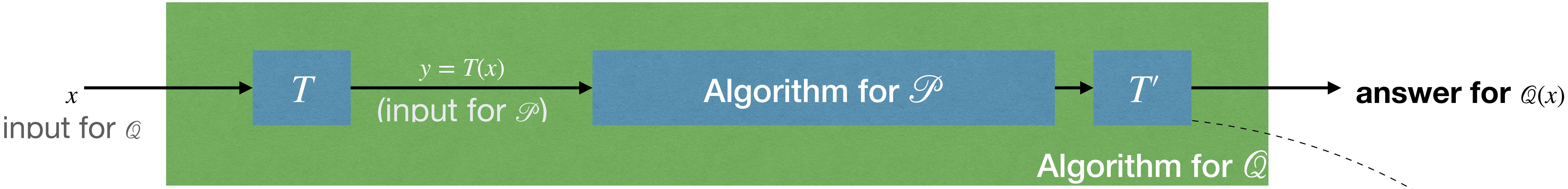
- If we have an algorithm for $\mathscr{P}$, and can convert an instance of $\mathscr{Q}$ to an instance of $\mathscr{P}$, then we effectively have an algorithm for $\mathscr{Q}$ already!



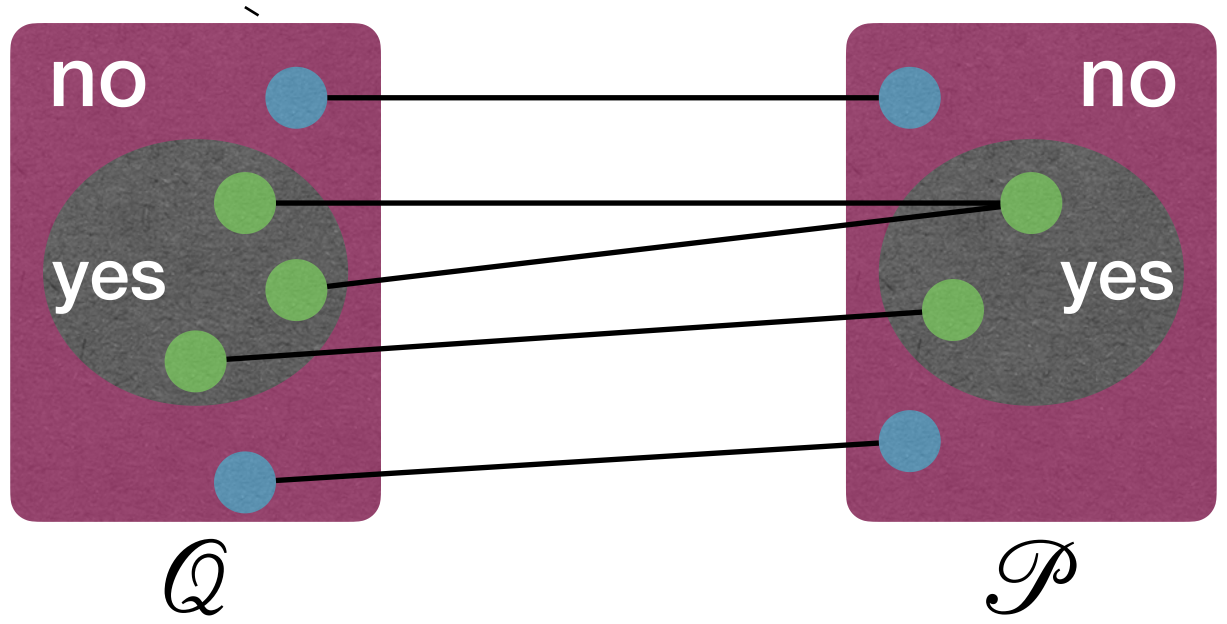
- Example: shortest distances in unit-length graphs via BFS

Polynomial Reduction

- Define function T: input of decision problem $\mathscr{Q} \rightarrow$ input of decision problem $\mathscr{P}$.
- T is a polynomial reduction from $\mathscr{Q}$ to $\mathscr{P}$ if
  - T can be computed within polynomial time (w.r.t. input length).
  - Input x is a “yes” input for $\mathscr{Q}$ **iff** $T(x)$ is a “yes” input for $\mathscr{P}$.

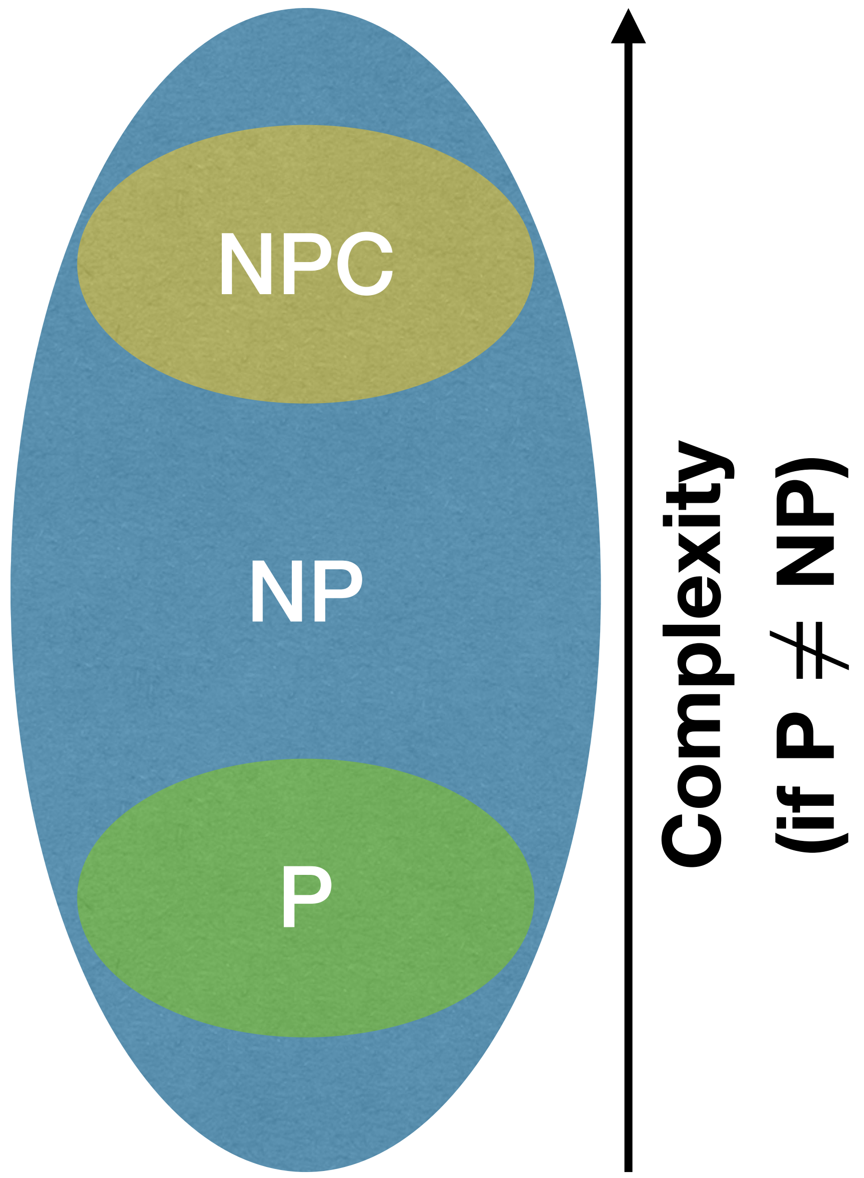


  - $\mathscr{Q}$ is polynomially reducible to $\mathscr{P}$: $\mathscr{Q} \leq_P \mathscr{P}$
  - $\mathscr{P}$ is at least as hard as $\mathscr{Q}$.



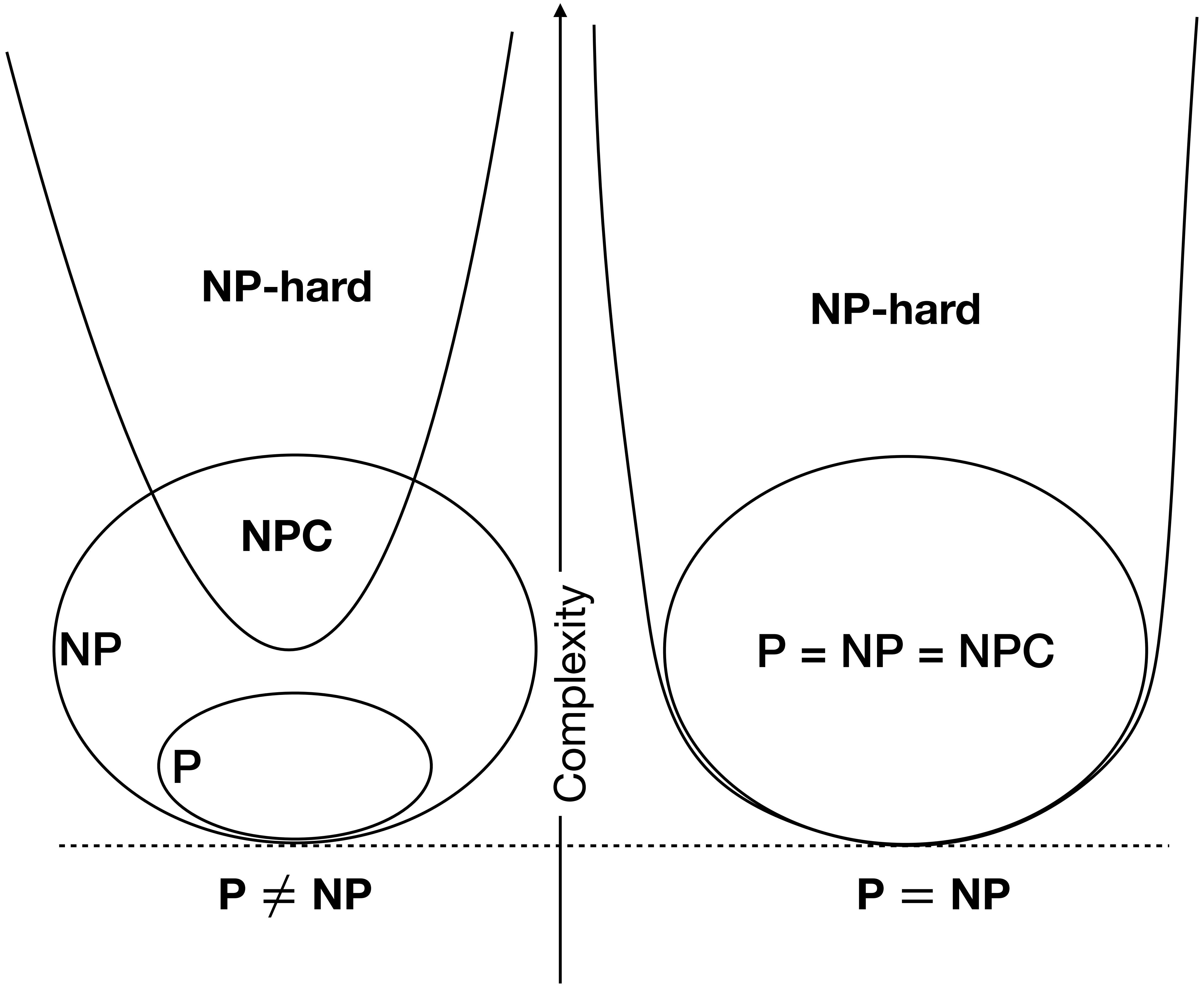
The hardest among the hard ones

- NP-Complete (**NPC**) problems are the **hardest** ones in NP.
- A decision problem $\mathscr{P}$ is **NPC** if:
  - The problem $\mathscr{P}$ is in **NP**.
  - If we have an algorithm for $\mathscr{P}$, then **all** problems in **NP** can be solved with limited extra work.
  - For every problem $\mathscr{Q}$ in **NP**, it is polynomially reducible to $\mathscr{P}$.



NPC and NP-hard

- A decision problem $\mathscr{P}$ is **NP**-hard if:
  - For every problem $\mathscr{Q}$ in **NP**, it is polynomially reducible to $\mathscr{P}$.
- **NP**-hard problems are the ones that are "at least as hard as the hardest problems in **NP**".
- A decision problem $\mathscr{P}$ is **NPC** if it is both **NP** and **NP**-hard



Prove a decision problem is NPC

- How to prove a decision problem is **NPC**?
  - Show the problem is in **NP**.
  - Show every $\mathcal{Q} \in \mathbf{NP}$ is polynomially reducible to the problem.

Infinity!

SAT: the First NPC Problem

- **SAT**: Given a Boolean formula ϕ in CNF, is ϕ satisfiable?
  - Example:
    $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \overline{x_1}) \wedge (x_2 \vee \overline{x_1} \vee \overline{x_2}) \wedge (x_4)$
- The Cook-Levin Theorem: SAT is NP-Complete.
  - [Western world] Stephen Cook, 1971.
  - [USSR] Leonid Levin, 1973.

Stephen Cook

Leonid Levin

And it all starts here…

- Once we find the first **NPC** problem, finding other **NPC** problems will be much easier:
  - Show the candidate **P** ∈ **NP**.
  - Show SAT (or other **NPC** problem) is polynomially reducible to.

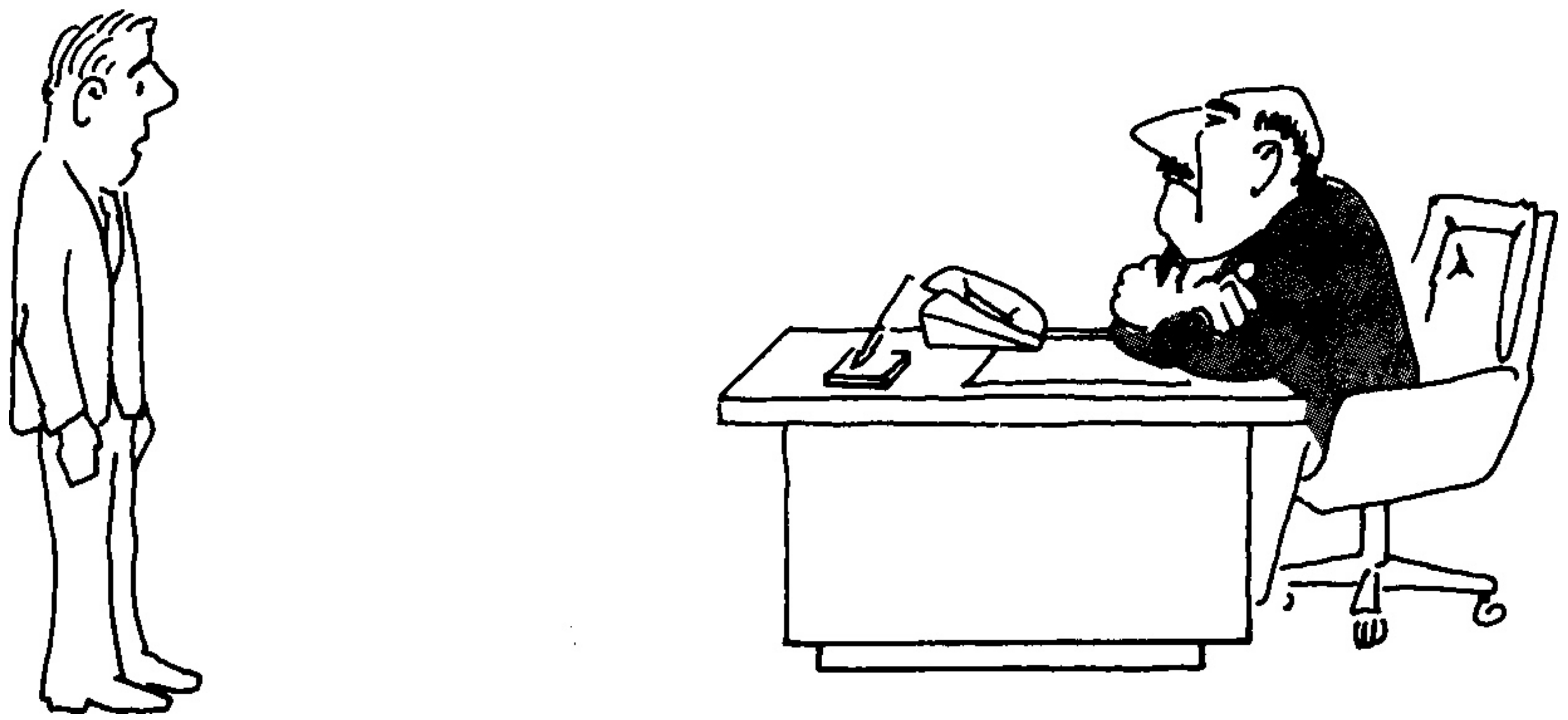
Beware of the direction of reduction!

- Leveraging the Cook-Levin Theorem, Richard Karp lists 21 **NPC** problems, in the year of 1972.
- More **NPC** problems are later found… (e.g., problems in the book [Garey & Johnson])

Richard Karp

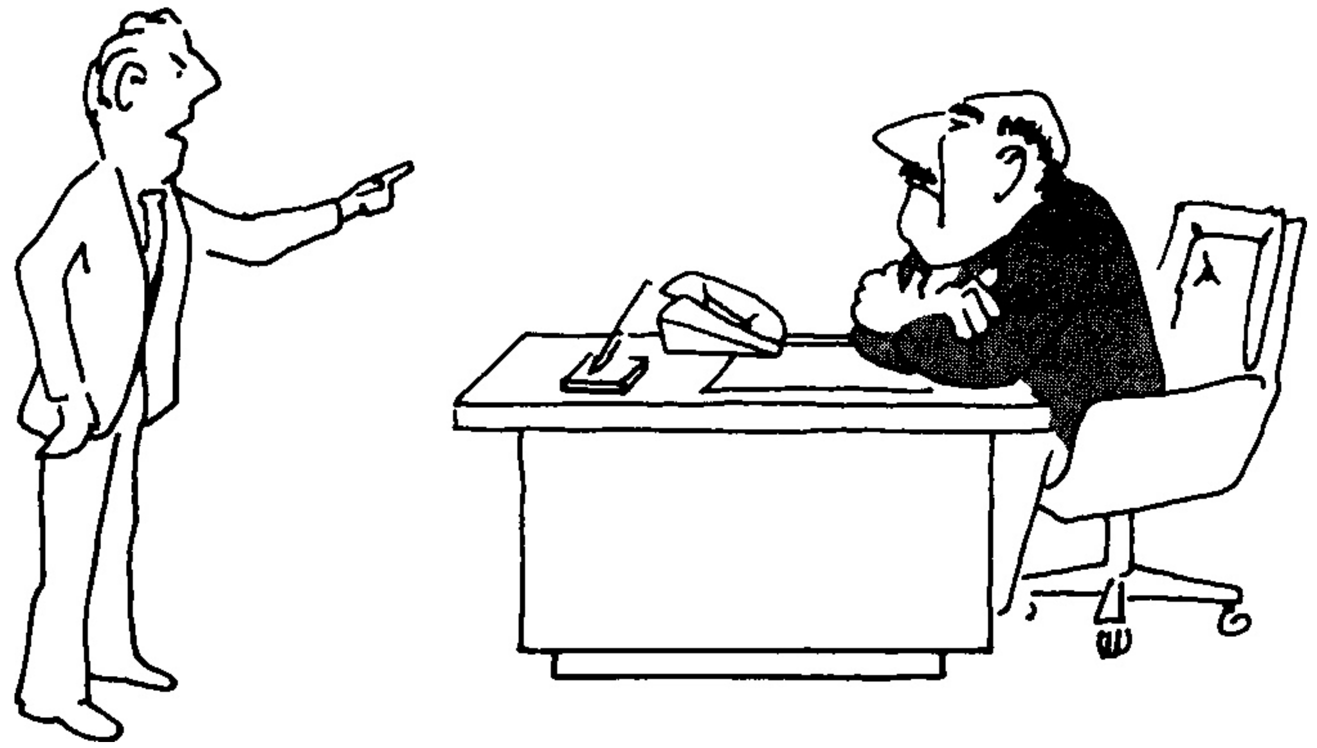
Saving your job…

when you don’t know complexity theory



“I can’t find an efficient algorithm, I guess I’m just too dumb.”

when you have a lower bound



“I can’t find an efficient algorithm, because no such algorithm is possible!”

when you find it is NP-Complete

“I can’t find an efficient algorithm, but neither can all these famous people.”

3-SAT is NPC

- 3-SAT: given a Boolean formula ϕ in CNF in which each clause has **exactly three** distinct literals, is ϕ satisfiable?
  - Example: $\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \overline{x_4} \vee x_1) \wedge (x_1 \vee \overline{x_1} \vee \overline{x_2})$
- The easy part: 3-SAT is in **NP**.
  - Any valid truth assignment can be a certificate.
  - So "yes" instances can be verified in polynomial time.
- The more challenging part: 3-SAT is **NP**-hard.
  - Reduce 3-SAT to SAT? (Show 3-SAT $\leq_P$ SAT?)
  - Reduce SAT to 3-SAT. (Show SAT $\leq_P$ 3-SAT.)

3-SAT is NP-hard

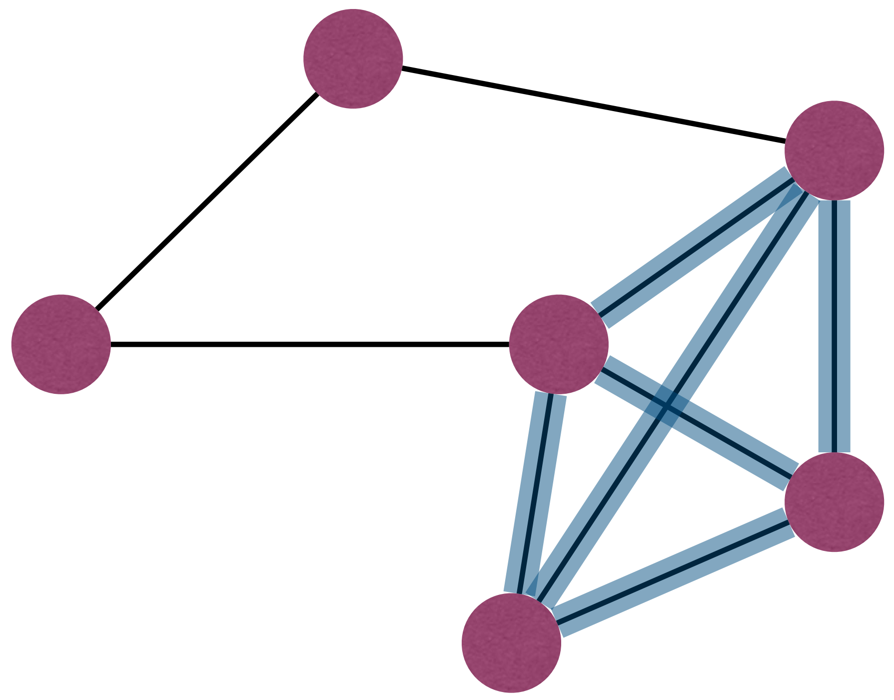
- Reduce SAT to 3-SAT. (Show SAT $\leq_P$ 3-SAT.)
- Convert an instance ϕ of SAT to an instance ϕ' of 3-SAT:
  - Conversion can be done in polynomial time (w.r.t. $|\phi|$).
  - ϕ is satisfiable iff ϕ' is satisfiable.

3-SAT is NP-hard

- Convert each clause C of ϕ in the following way:
  - $C = (z_1)$, let
    $C' = (x_1 \vee x_2 \vee z_1) \wedge (x_1 \vee \overline{x_2} \vee z_1) \wedge (\overline{x_1} \vee x_2 \vee z_1) \wedge (\overline{x_1} \vee \overline{x_2} \vee z_1)$
  - $C = (z_1 \vee z_2)$, let $C' = (x_1 \vee z_1 \vee z_2) \wedge (\overline{x_1} \vee z_1 \vee z_1)$
  - $C = (z_1 \vee z_2 \vee z_3)$, simply let $C' = C$
  - $C = (z_1 \vee z_2 \vee \ldots \vee z_k)$, where $k > 3$, let
    $C' = (z_1 \vee z_2 \vee x_1) \wedge (\overline{x_1} \vee z_3 \vee x_2) \wedge (\overline{x_2} \vee z_4 \vee x_3) \wedge \ldots$
    $\wedge (\overline{x_{k-4}} \vee z_{k-2} \vee x_{k-3}) \wedge (\overline{x_{k-3}} \vee z_{k-1} \vee x_k)$

Clique is NPC

- Clique: Given (G, k), does graph G contain clique of size k?
- The easy part: Clique is in **NP**.
  - Any k vertices in a clique can be a certificate.
  - So "yes" instances can be verified in polynomial time.
- The more challenging part: Clique is **NP**-hard.
  - Show 3-SAT $\leq_P$ Clique.



The vertices connected by blue edges
(pairwise adjacent) are 4-clique

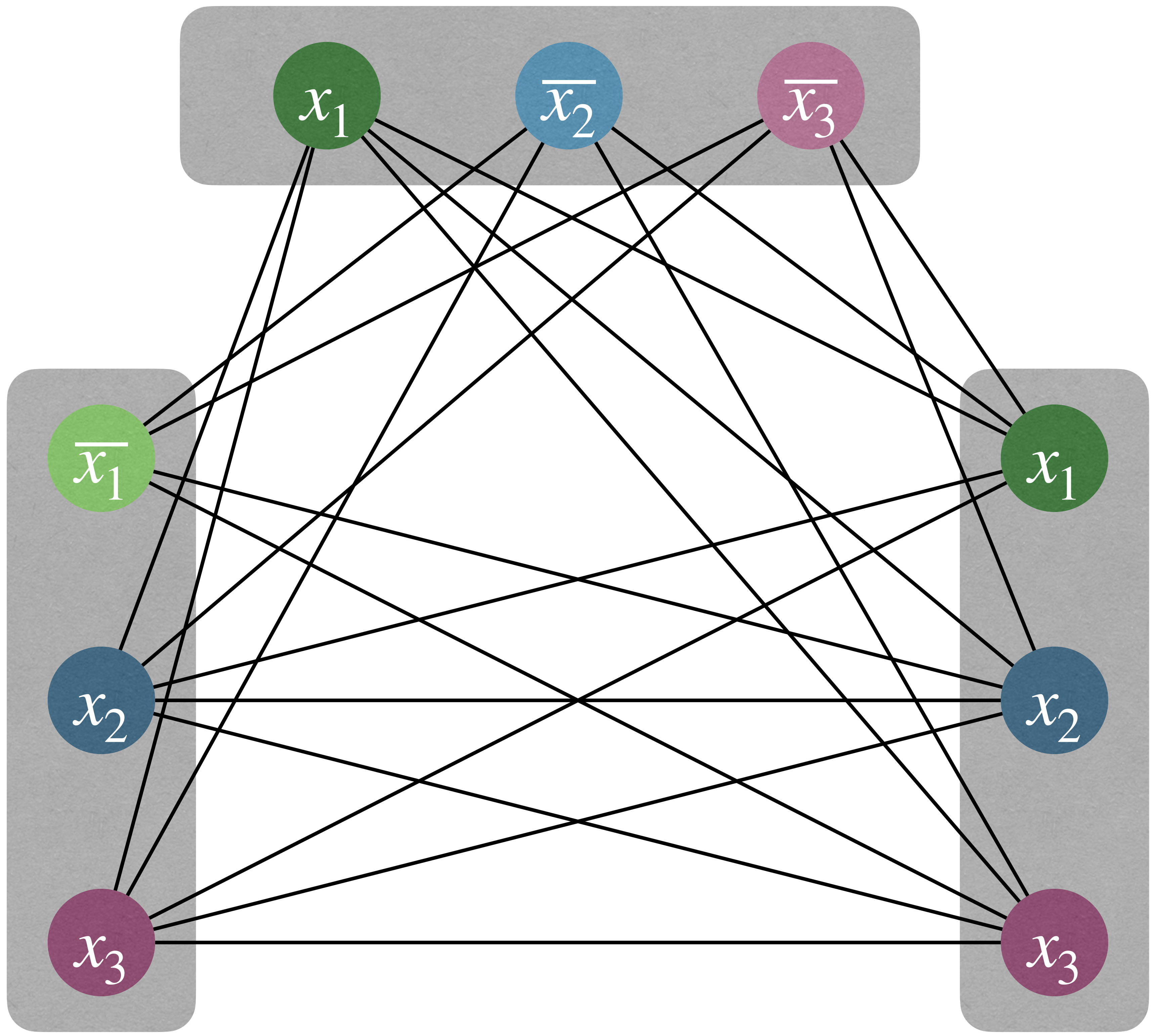
There are 6 1-cliques (all these vertices)
There are 9 2-cliques (all these edges)
There are 4 3-cliques (triangles in the 4-clique)

Clique is NP-hard

- Show 3-SAT $\leq_P$ Clique
  - Given an instance ϕ of 3-SAT, convert it to an instance (G, k) of Clique within polynomial time.
  - Answer for ϕ of 3-SAT is YES iff answer for (G, k) of Clique is YES.
- Conversion procedure:
  - Let k be the number of clauses in ϕ.
  - For each clause C_i of ϕ create three nodes $v_{i,1}, v_{i,2}, v_{i,3}$.
  - Connect two nodes $v_{i,j}$ and $v_{i',j'}$ iff: $i \neq i'$, and $v_{i,j}$ and $v_{i',j'}$ are not literals negating each other.

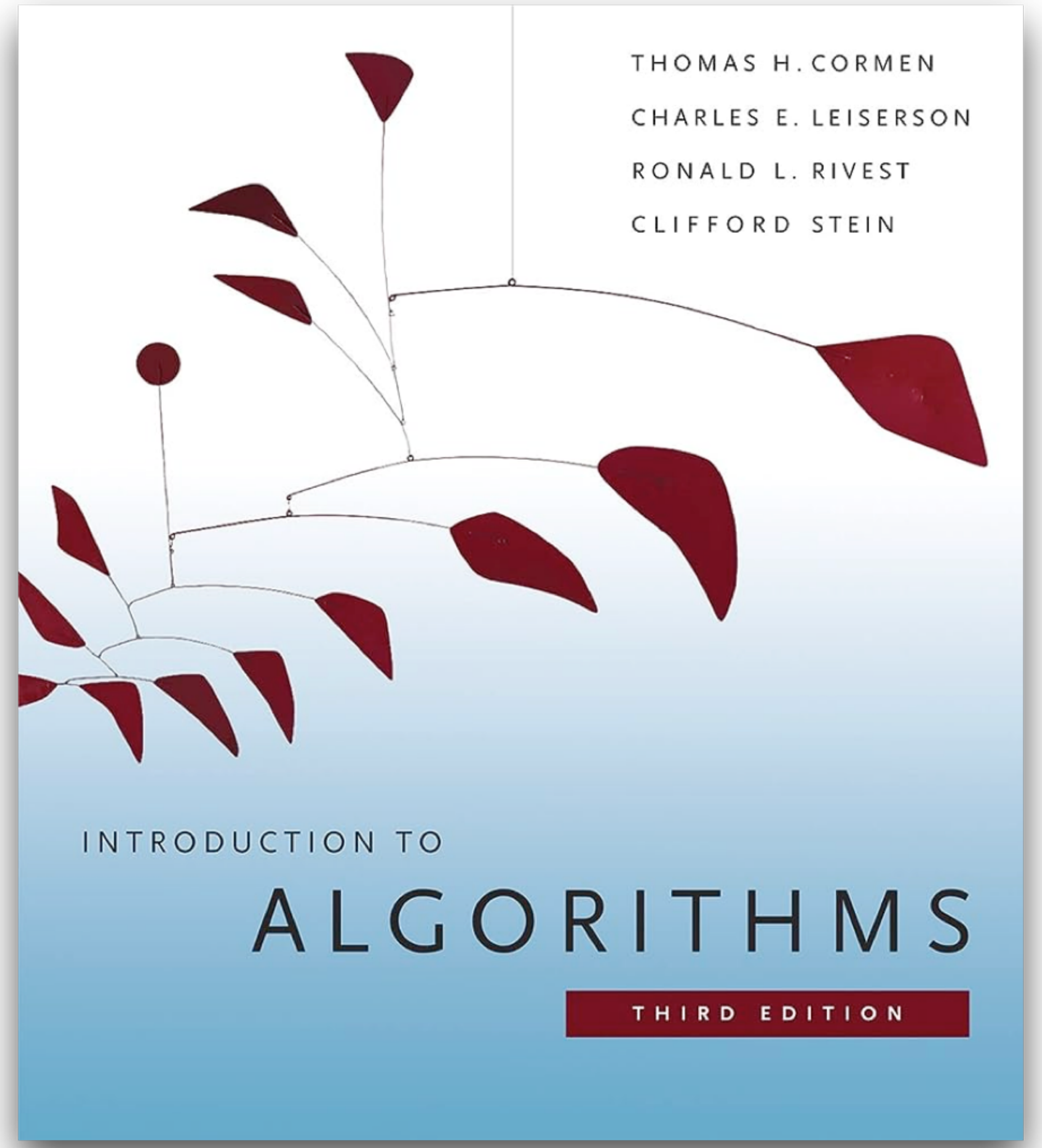
Clique is NP-hard

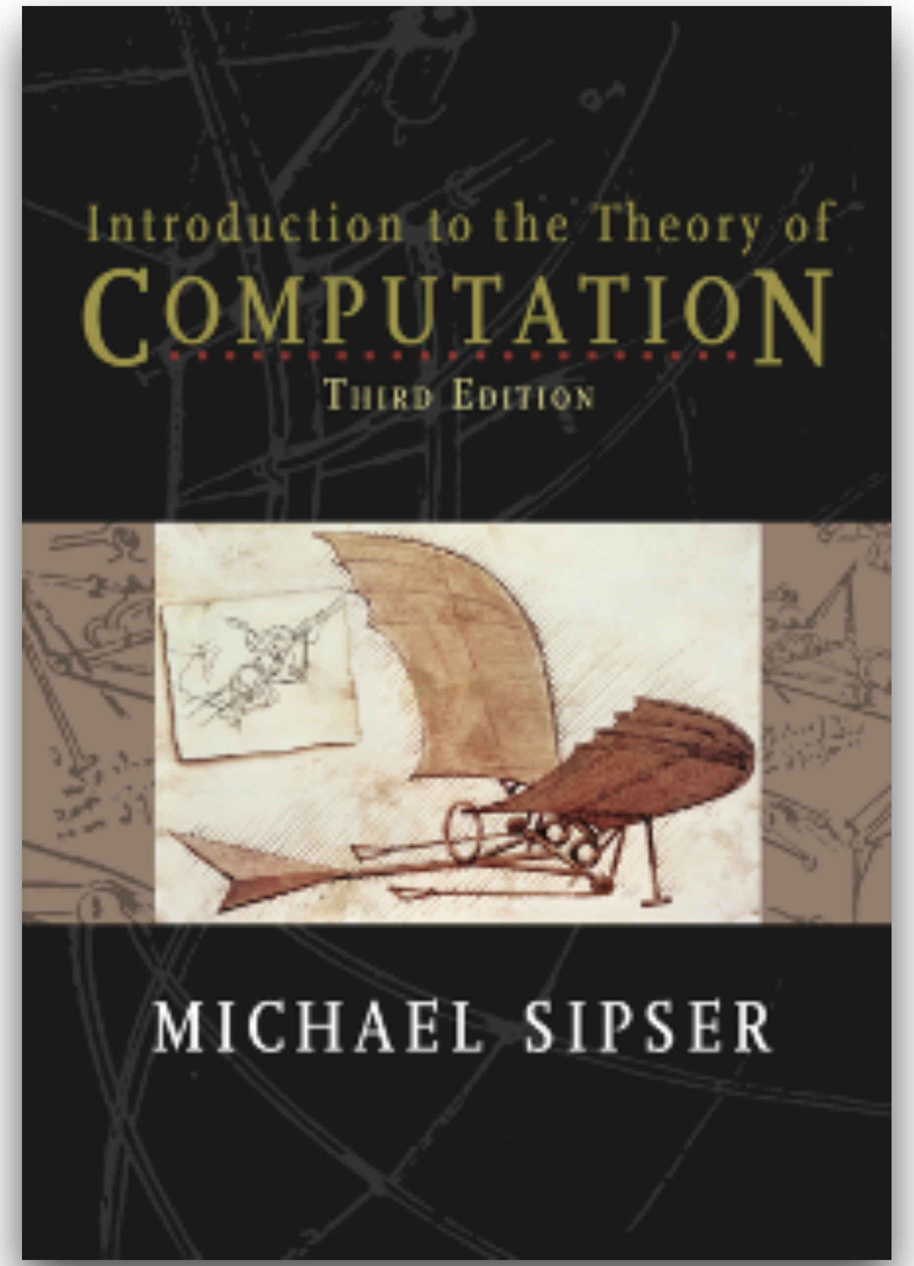
- $\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
  - $\Longrightarrow$ If ϕ is satisfiable, in each clause at least one literal will be satisfied. Nodes correspond to these k literals will be a clique.
  - $\Longleftarrow$If there is a k clique in the graph, this clique will contain one node from each clause. These nodes correspond to non-conflicting literals, implying a satisfying assignment.

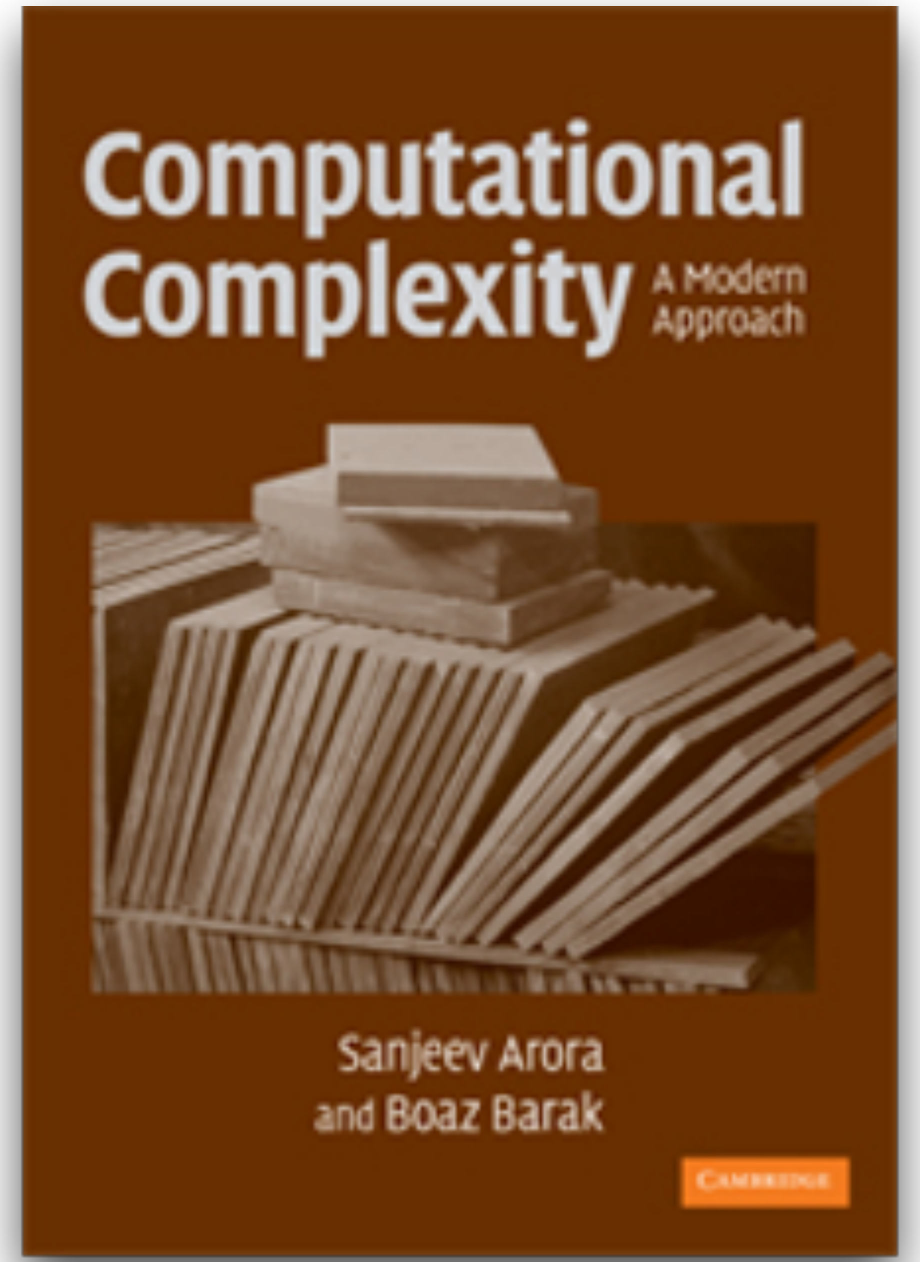


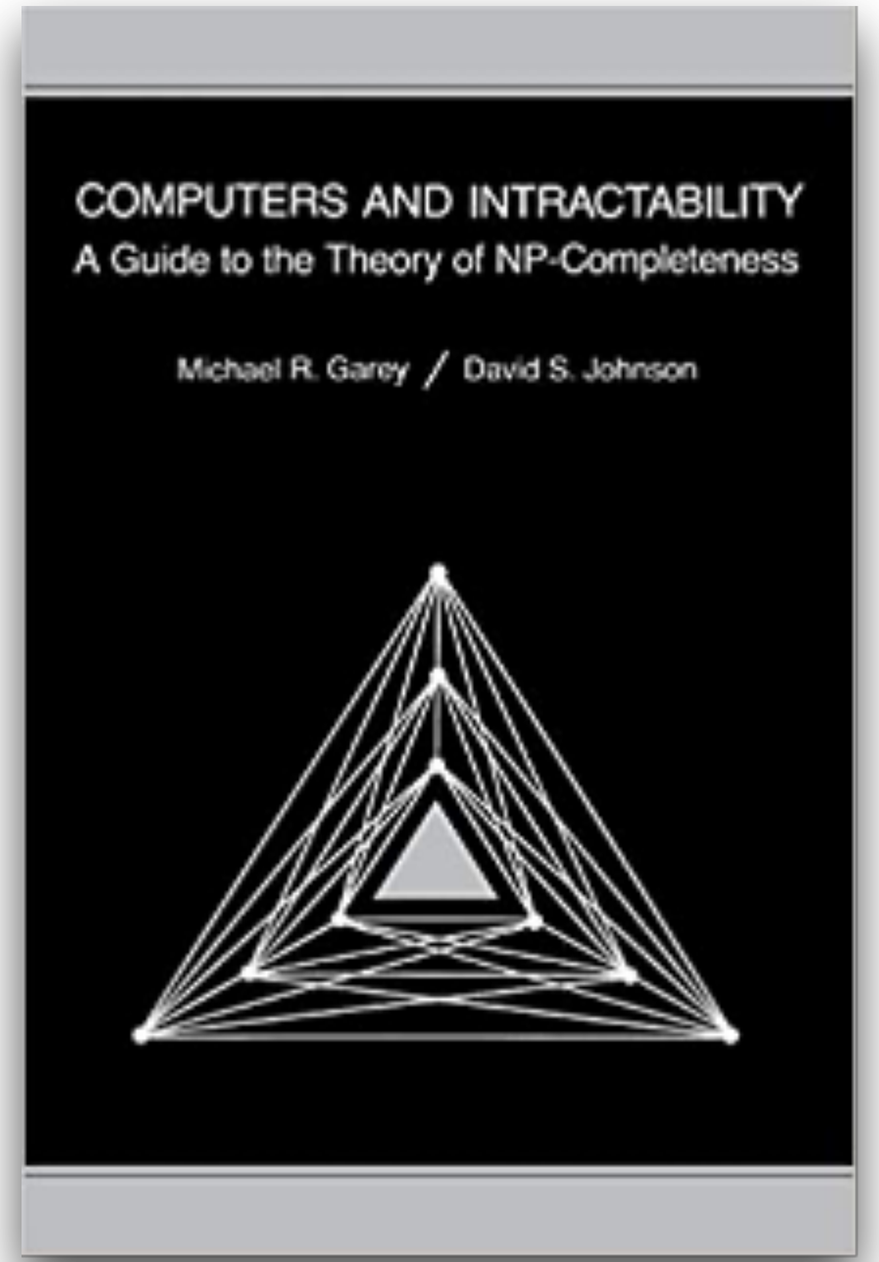
Further reading

- [CLRS] Ch.34 (34.1-34.5)









Refer to [Sipser] and [Arora & Barak] for more about computational complexity

Refer to [Garey & Johnson] for more NP-completeness problems