Complexity Theory

Birgit Vogtenhuber

Outline

- Motivation
- Two introductory examples
- Complexity classes
- Polynomial time reductions
- **NP**-completeness
- **P** vs. **NP**

Motivation

- So far in this course we've seen a lot of **good news**: problems that can be solved quickly
  - in close to linear time (minimum spanning tree, convex hull, . . .)
  - in time that is some small polynomial function of the input size (minimum weight triangulation, all shortest paths, . . .)
- The topic of today is a form of **bad news**: evidence that there are many important problems which can't be solved quickly.
- **Complexity theory**: dedicated to classifying problems by how "hard" they are.

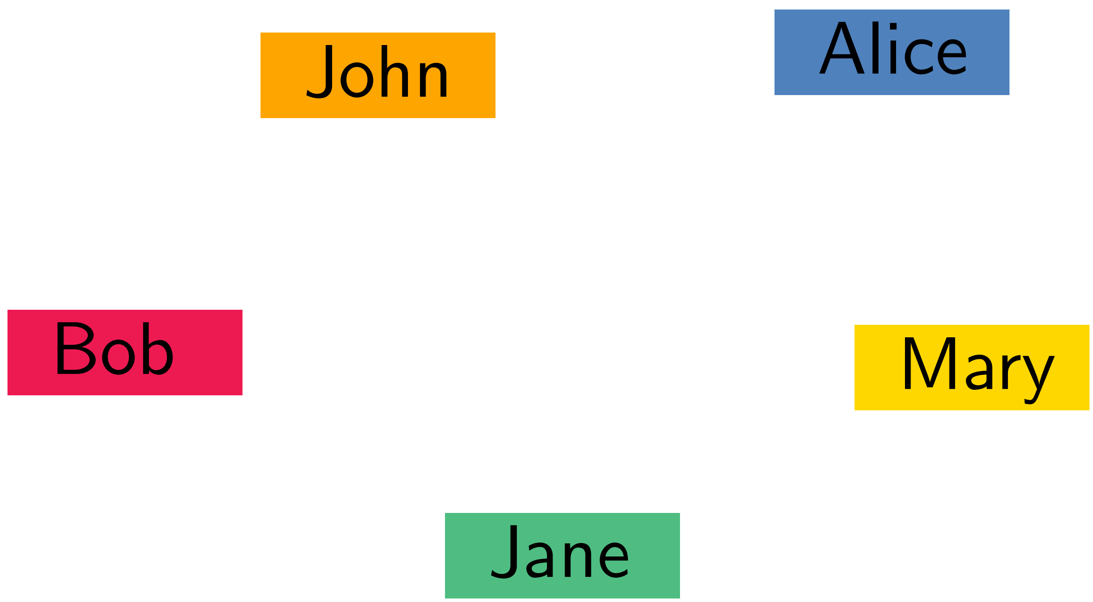
Why should we care?

- "Hard" problems come up all the time.
- Knowing they're hard lets you stop beating your head against a wall trying to solve them efficiently, and do something better:
  - Use a heuristic.
  - Solve the problem approximately instead of exactly.
  - Use an exponential time solution anyway.
  - Choose a better abstraction.

First: Let's look at some examples ...

Dinner Party

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along well.



Dinner Party

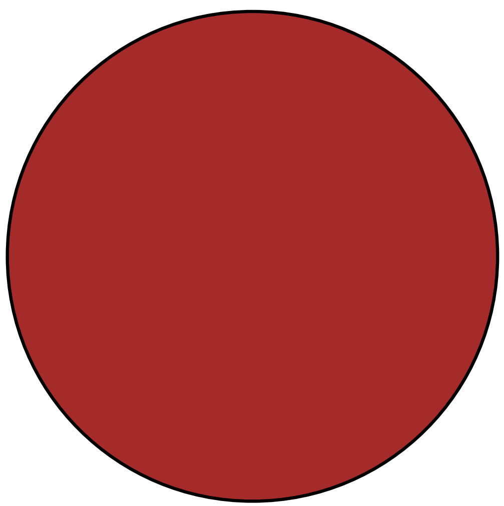
- **Problem:** Seat all guests around a table, so that people who sit next to each other get along well.

	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	

Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

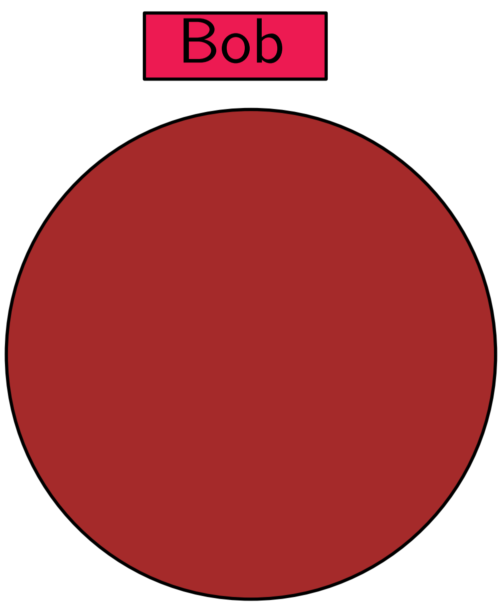
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

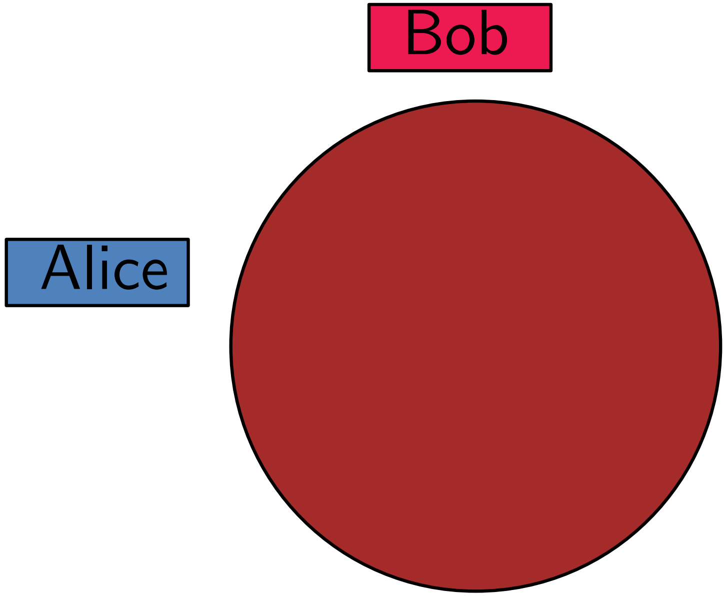
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

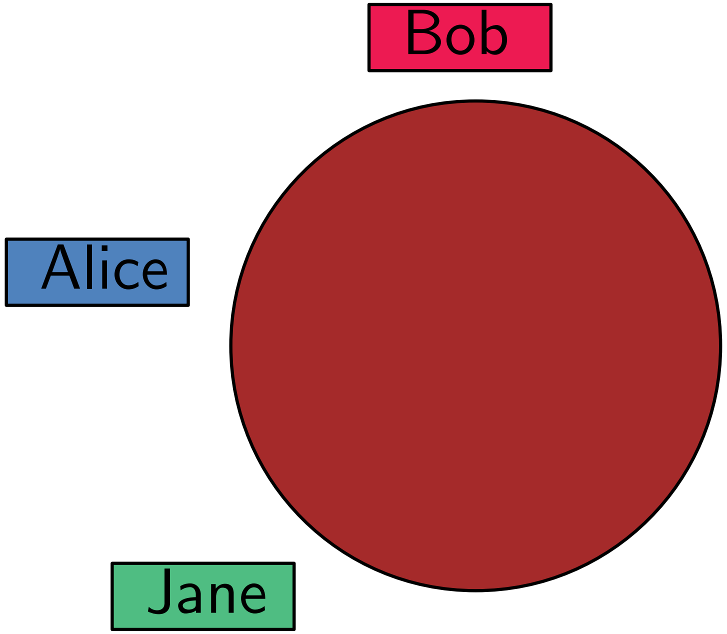
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

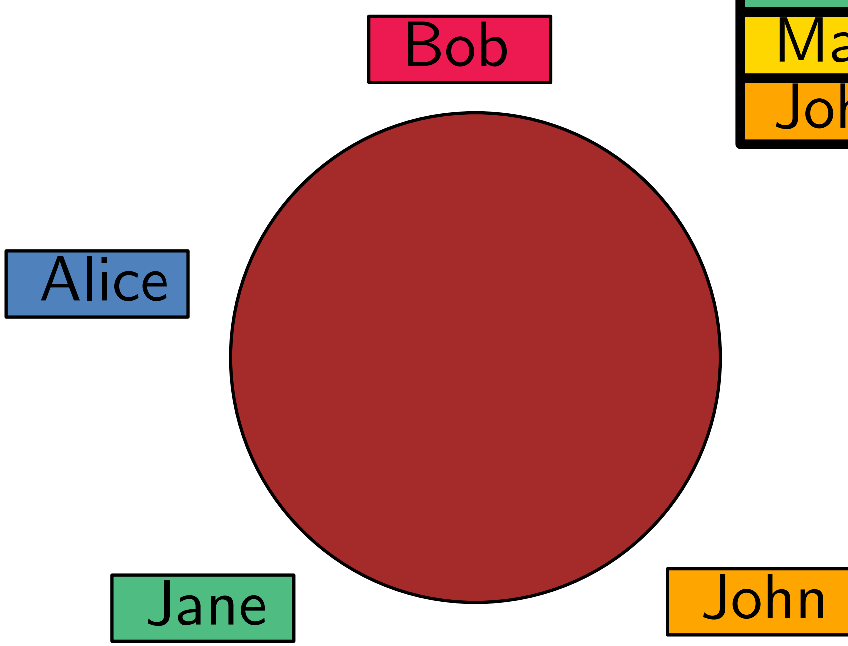
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

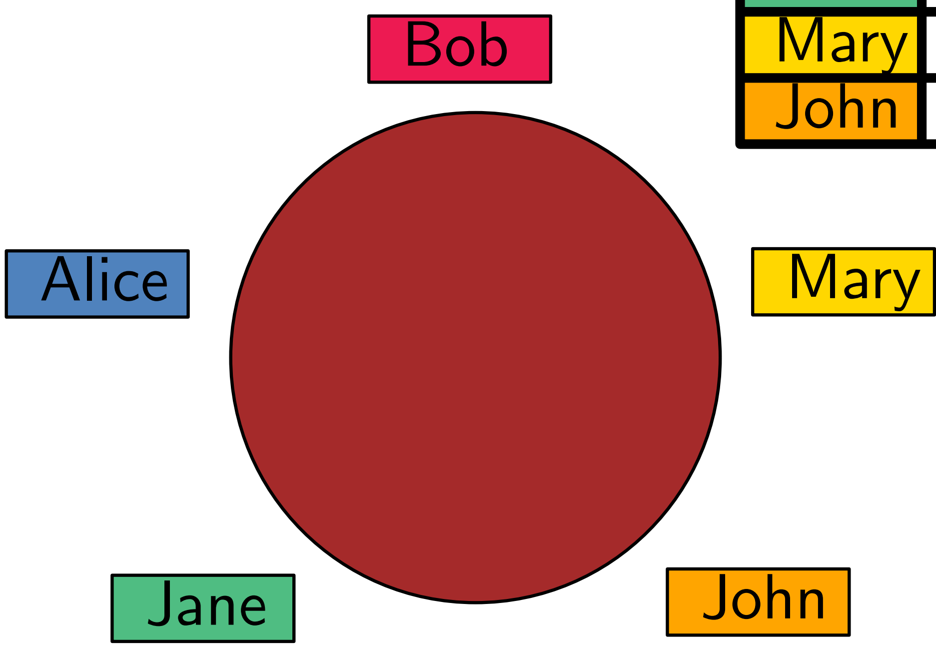
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

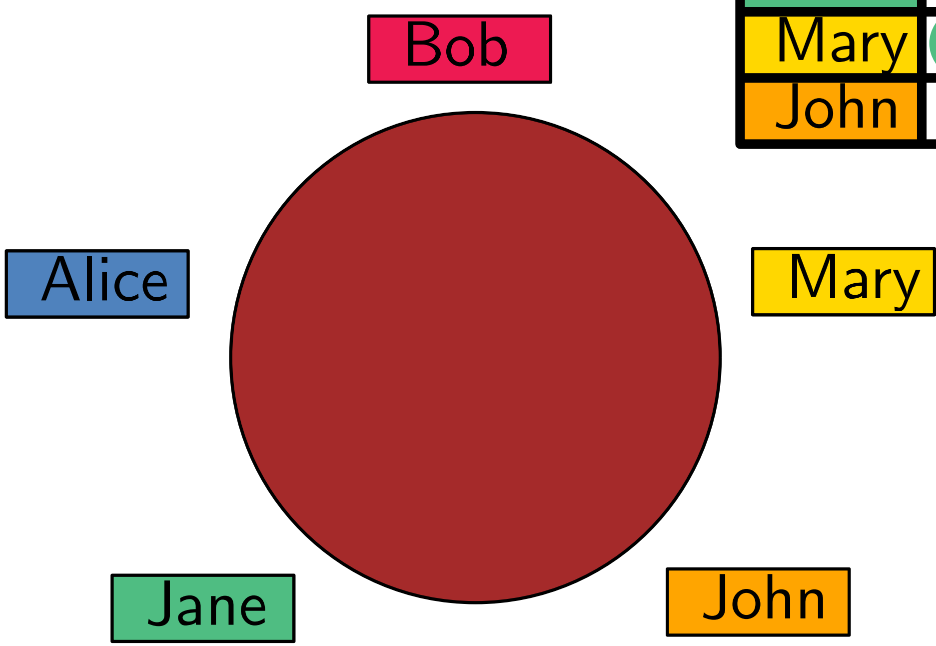
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Example Solution

- **Problem:** Seat all guests around a table, so that people who sit next to each other get along.

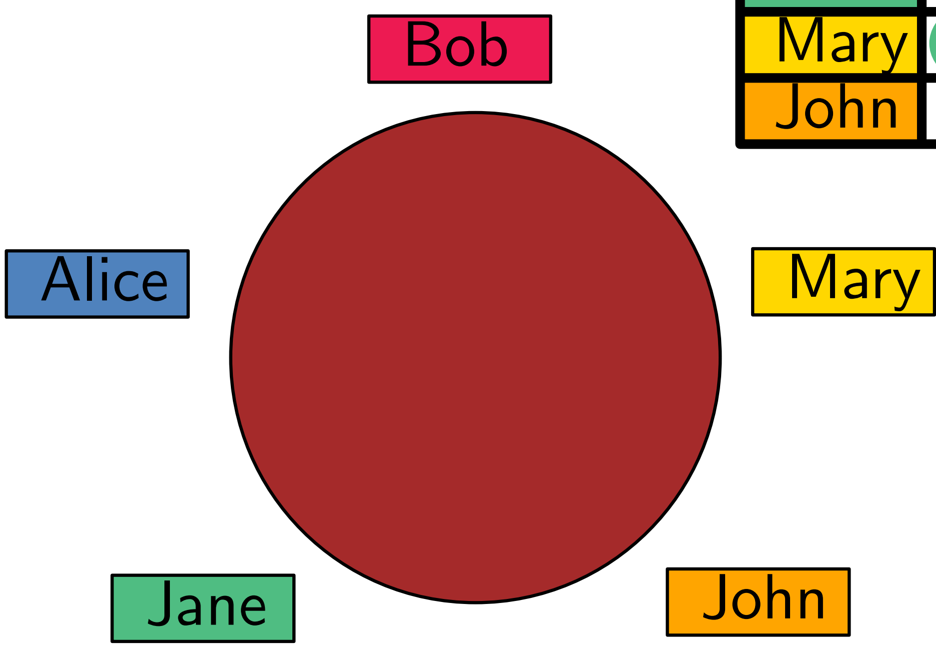
	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Naive Algorithm

- **Observation:** Given a seating order, one can efficiently verify if all guests get along well with their neighbors.

	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		✓
Jane		✓			✓
Mary	✓	✓	✓		✓
John			✓	✓	



Dinner Party: Naive Algorithm

- **Observation:** Given a seating order, one can efficiently verify if all guests get along with their neighbors.
- direct **Problem solution:**
  - Verify **each seating arrangement** of the guests.
  - Stop if seating arrangement fine for all guests.
  - Stop if no seating arrangement left to verify.
- **How many steps** in the worst case for n guests?
  - $(n-1)!/2$ different seating orders:

n	5	15	100
$(n-1)!/2$	12	43589145600	$\approx 4.5 \cdot 10^{155}$

Dinner Party: Naive Algorithm

- **Observation:** Given a seating order, one can efficiently verify if all guests get along with their neighbors.
- direct **Problem solution:**
  - Verify **each seat**
  - Stop if seating a
  - Stop if no seatin

computer with 10^{11} instructions per second $\Rightarrow$ $> 10^{137}$ years!

- **How many steps** in the worst case for n guests?
  - $(n-1)!/2$ different seating orders:

n	5	15	100
$(n-1)!/2$	12	43589145600	$\approx 4.5 \cdot 10^{155}$

Dinner Party: Naive Algorithm

- **Observation:** Given a seating order, one can efficiently verify if all guests get along with their neighbors.
- direct **Problem solution:**
  - Verify **each seat**
  - Stop if seating a
  - Stop if no seatin

D-Cluster Graz: $31 \cdot 10^{12} < 10^{14}$ instr./sec. $\Rightarrow$ still $> 10^{134}$ years!

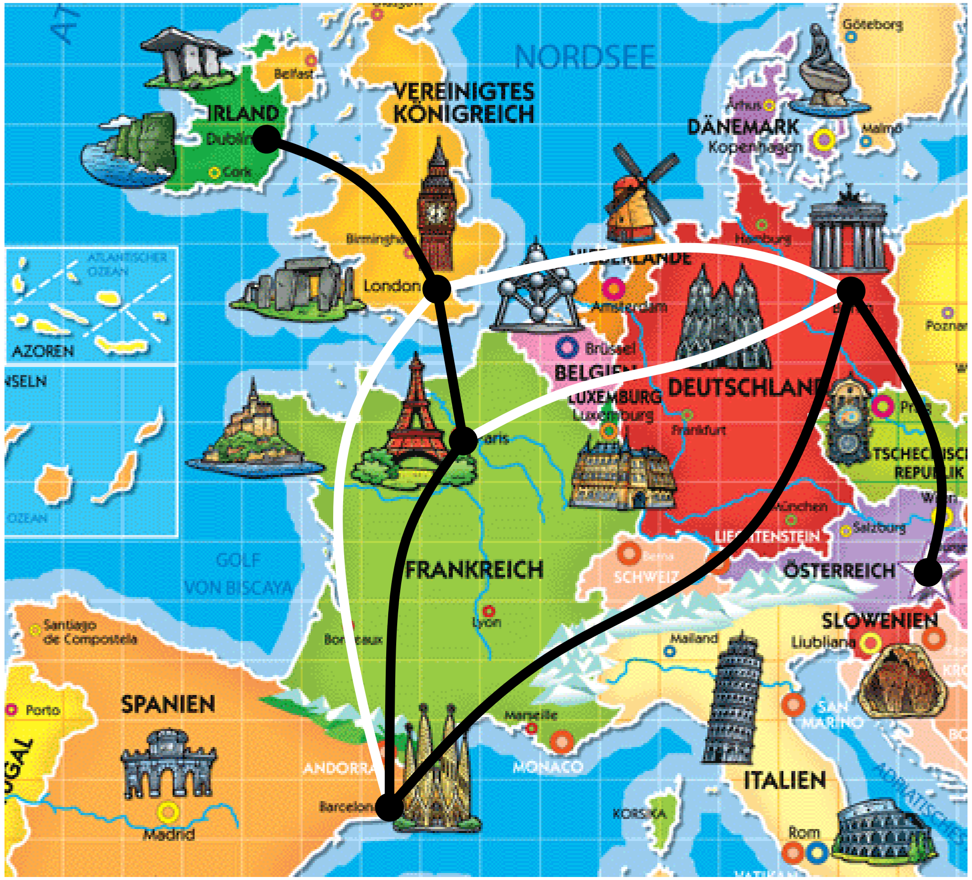
- ts?

Universe: < 14 Mrd. $= 1.4 \cdot 10^{10}$ years old

n	5	15	100
$(n-1)!/2$	12	43589145600	$\approx 4.5 \cdot 10^{155}$

City Tour: Example Solution

- **Problem:** Plan a trip that visits every location exactly once (only direct connections).



City Tour: Naive Algorithm

- **For every starting location:**
  - try all reachable sites not yet visited
  - backtrack and retry
  - repeat the process until stuck or done
- **How much time** for n cities?
  - up to $n!/2$ different orderings for the cities:

n	5	15	100
$n!/2$	60	653837184000	$\approx 4.5 \cdot 10^{157}$

Even worse than the previous problem!

City Tour: Variation

Question: Can you design an efficient algorithm for the tour problem if the sites' map contains no cycles?

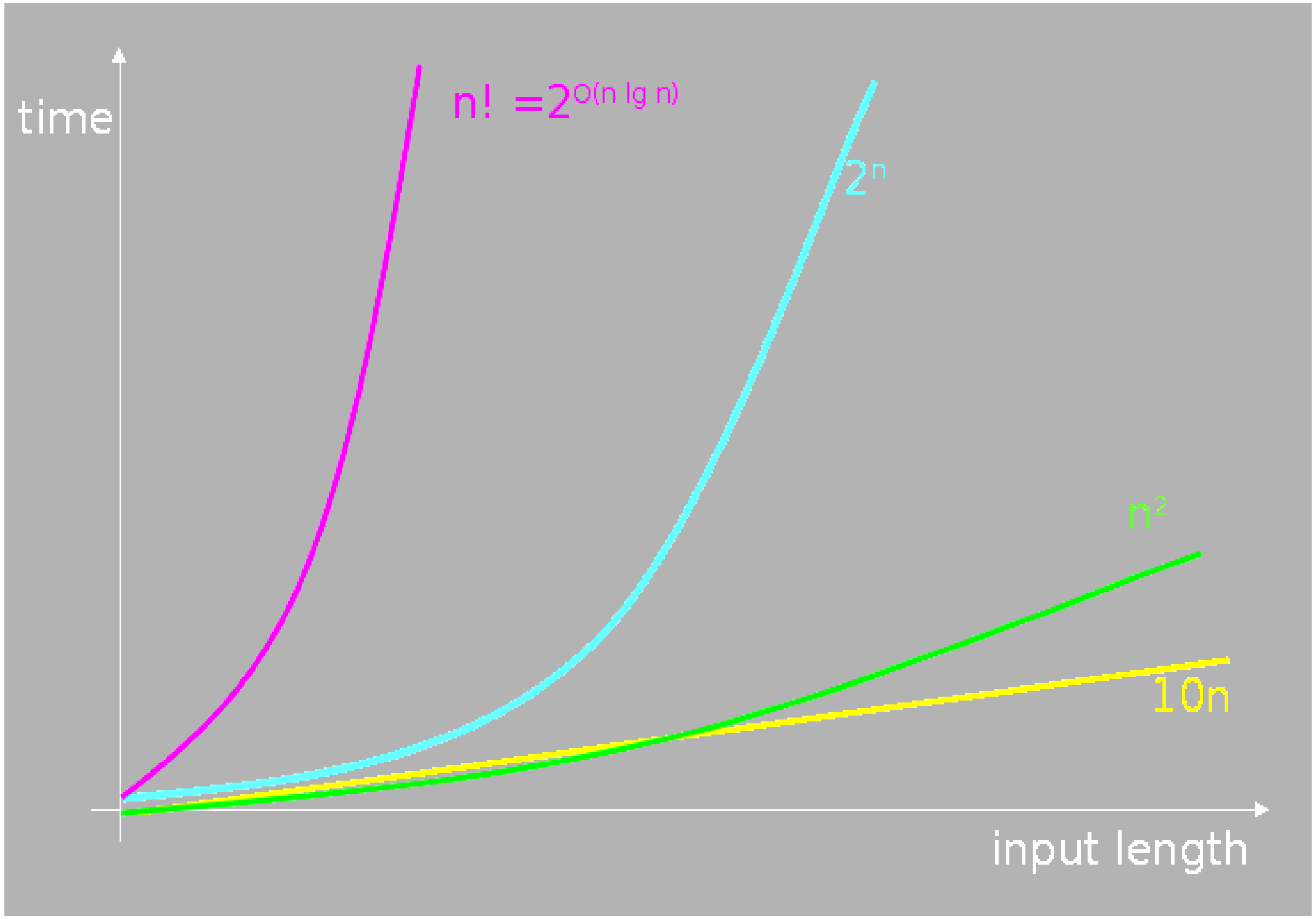
Tractability

Is a computational problem tractable?

- **YES!** and here is an **efficient algorithm** that solves it.
- **NO!** and I can **prove** it.
- **???** but what if **neither** is the case?

And what is "efficient"?

Growth Rate: Sketch



Tractability cont.

What is "efficient"?

In the context of **complexity theory**:

- **maybe reasonable:** at most polynomial $\equiv n^{O(1)}$
- **totally unreasonable:** exponential or more $\equiv 2^{n^{\Omega(1)}}$

Asymptotic notations: O, Ω, Θ, (o, ω)

Relations between Problems

City Tour vs. Dinner Party:
Is one fundamentally harder than the other?

- **Relations between problems:**
  - Assume that
    if there is an efficient algorithm for problem A
    then there is an efficient algorithm for problem B.

  $\Rightarrow$ B cannot be fundamentally harder than A.

- **Reducing B to A:** Make an efficient algorithm for B using the one from A. **Notation:** $B \leq_x A$

  type of reduction

  $\Rightarrow$ B cannot be fundamentally harder than A.
  $\Leftrightarrow$ A cannot be fundamentally easier than B.

Reducing City Tour to Dinner Party

City Tour vs. Dinner Party:
Is one fundamentally harder than the other?

- **First observation:** The problems are not so different:
  "… directly reachable from … " $\Leftrightarrow$ "… liked by …"

Really?

	Bob	Alice	Jane	Mary	John
Bob		✓		✓	
Alice	✓		✓		~~✓~~
Jane		✓			✓
Mary	✓	~~✓~~	~~✓~~		✓
John			✓	✓	

Reducing City Tour to Dinner Party

City Tour vs. Dinner Party:
Is one fundamentally harder than the other?

- **First observation:** The problems are not so different:
  "... directly reachable from ... " $\Leftrightarrow$ "... liked by ..."
- **Closing the cycle:** Tour only needs a path while Seating should produce a cycle.
  $\Rightarrow$ Invite an additional guest liked by everyone.
  $\Leftrightarrow$ Add a city that can be reached from everywhere.

Reducing City Tour to Dinner Party

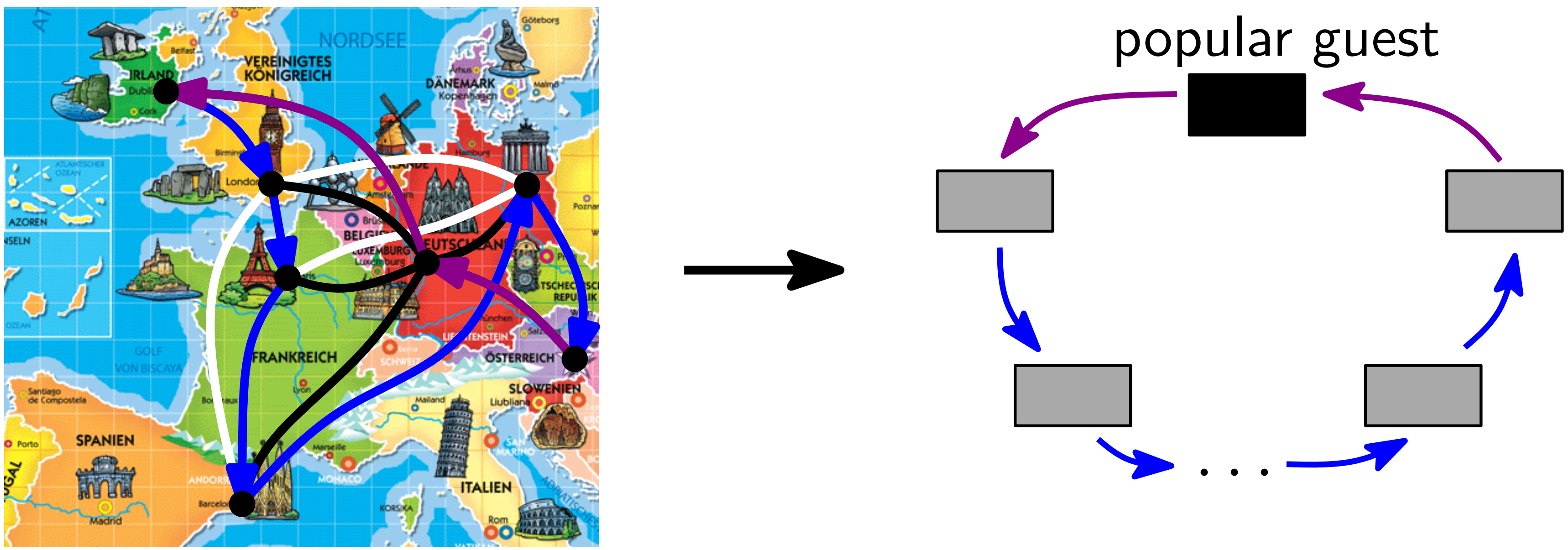
City Tour vs. Dinner Party:
Is one fundamentally harder than the other?

Reducing City Tour to Dinner Party

City Tour vs. Dinner Party:

Is one fundamentally harder than the other?

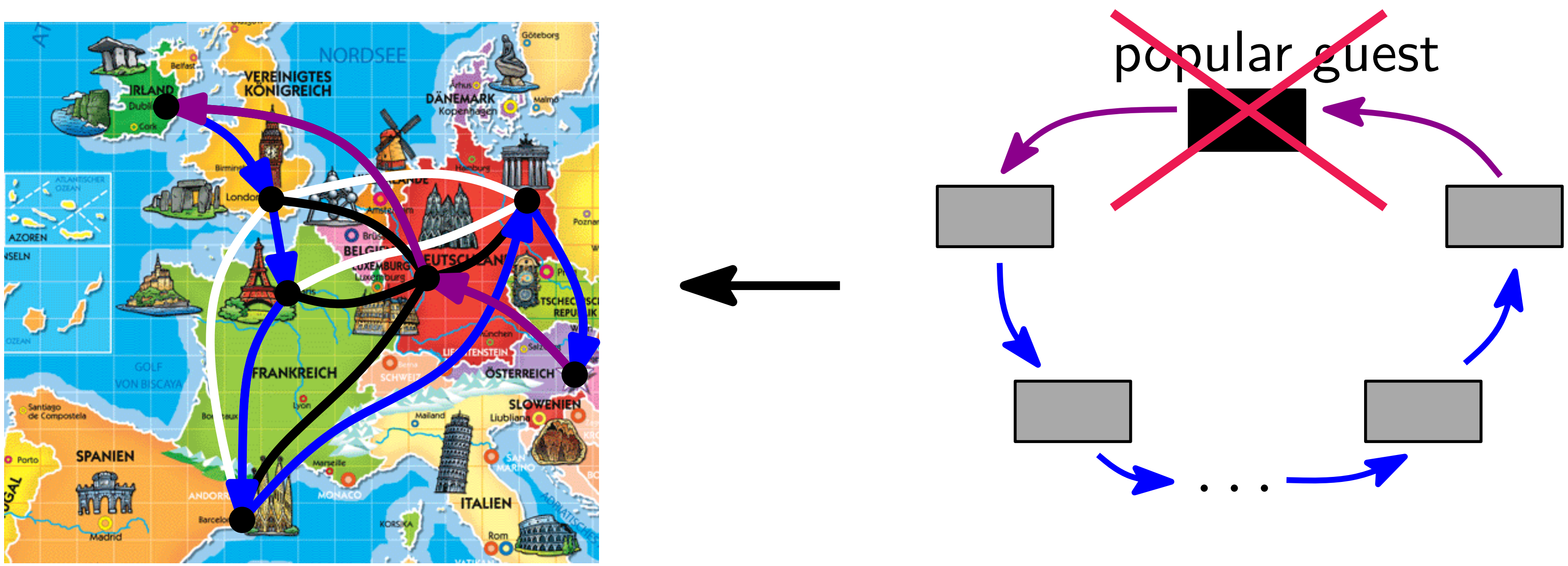
$\Rightarrow$ If there is a tour, there is also a way to seat all the imagined guests around the table.



Reducing City Tour to Dinner Party

City Tour vs. Dinner Party:
Is one fundamentally harder than the other?

$\Rightarrow$ If there is a seating, we can easily find a tour path (no tour $\Rightarrow$ no seating).



$\Rightarrow$ **City Tour $\leq_x$ Dinner Party:**
The seating problem is at least as hard as the tour problem.

Discussion

So Far ...

- We couldn't find efficient algorithms for the problems,
- nor prove they don't have one.
- But we managed to show a very powerful claim regarding the relation between their hardness.

Next ...

- Interestingly, one can also reduce the dinner party problem to the city tour problem. $\Rightarrow$ **Question:** Can you?
- Furthermore, there is a whole class of problems, which can be **pair-wise efficiently reduced** to each other.
- Before that: problems and complexity classes.

Classification of Problems

- There are many different complexity classes.
  We will only consider some of the most common.
- Technical point: many classes are defined in terms of **decision problems**, that is, problems of the type

  Does a certain structure exist?

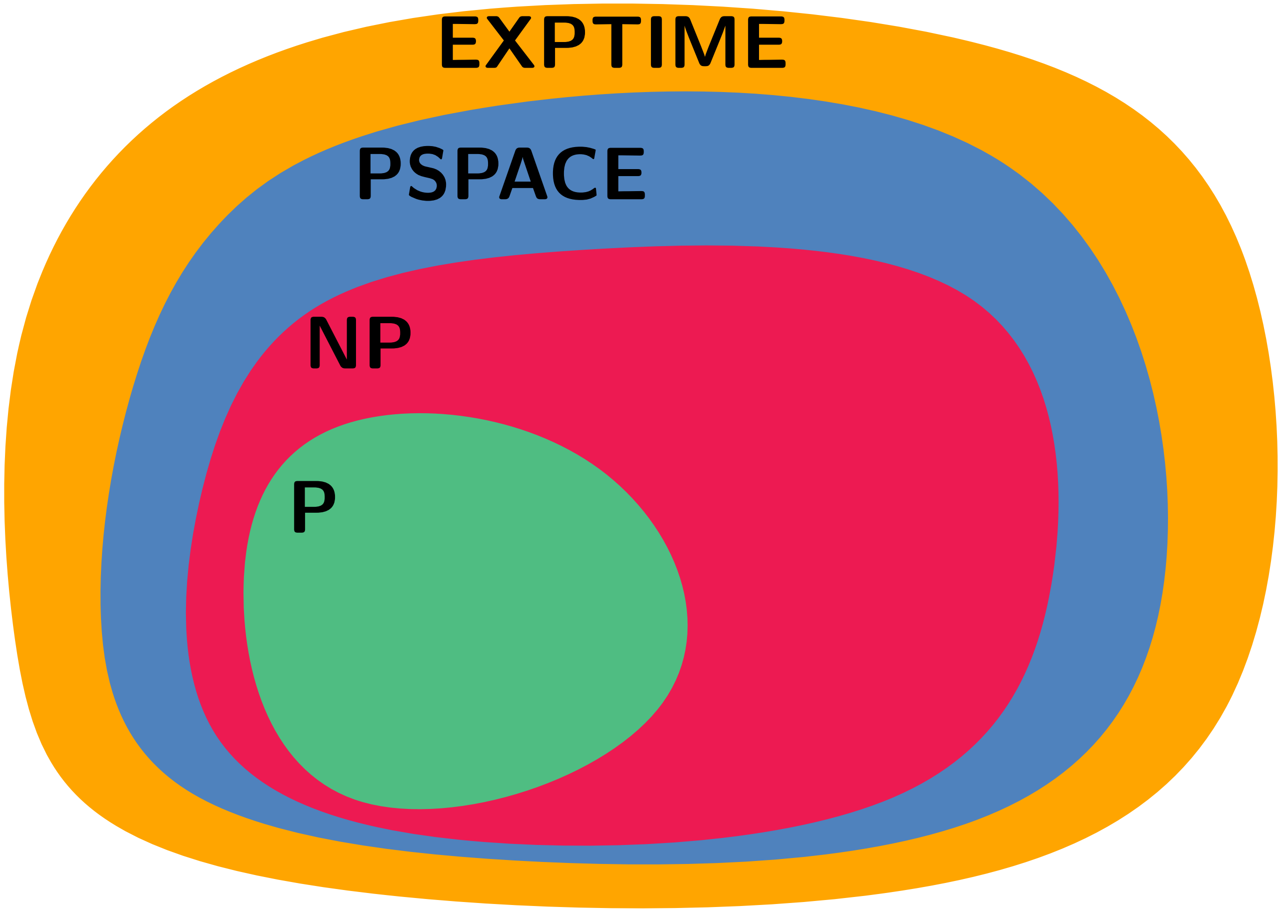
  rather than *How do I find the structure?*
- Example Tour: *Given a graph does there exist a tour that visits each vertex exactly once?*
- Example shortest paths: *Given a graph, does there exist a path from vertex u to vertex v with at most k edges?*

Some Complexity Classes

- **P**: Decision problems for which the answer is computable in **polynomial time**.
- **NP**: Decision problems for which a **positive answer** is **efficiently verifiable** via a "proof" (e.g., a solution).

  NP stands for **nondeterministic polynomial time**. (Attention: <u>not</u> for "non-polynomial").
- **PSPACE**: Decision problems for which the answer is computable unsing most a **polynomial amount of memory**, without worrying about how much time the decision takes.
- **EXPTIME**: Decision problems for which the answer is computable in **exponential time**.

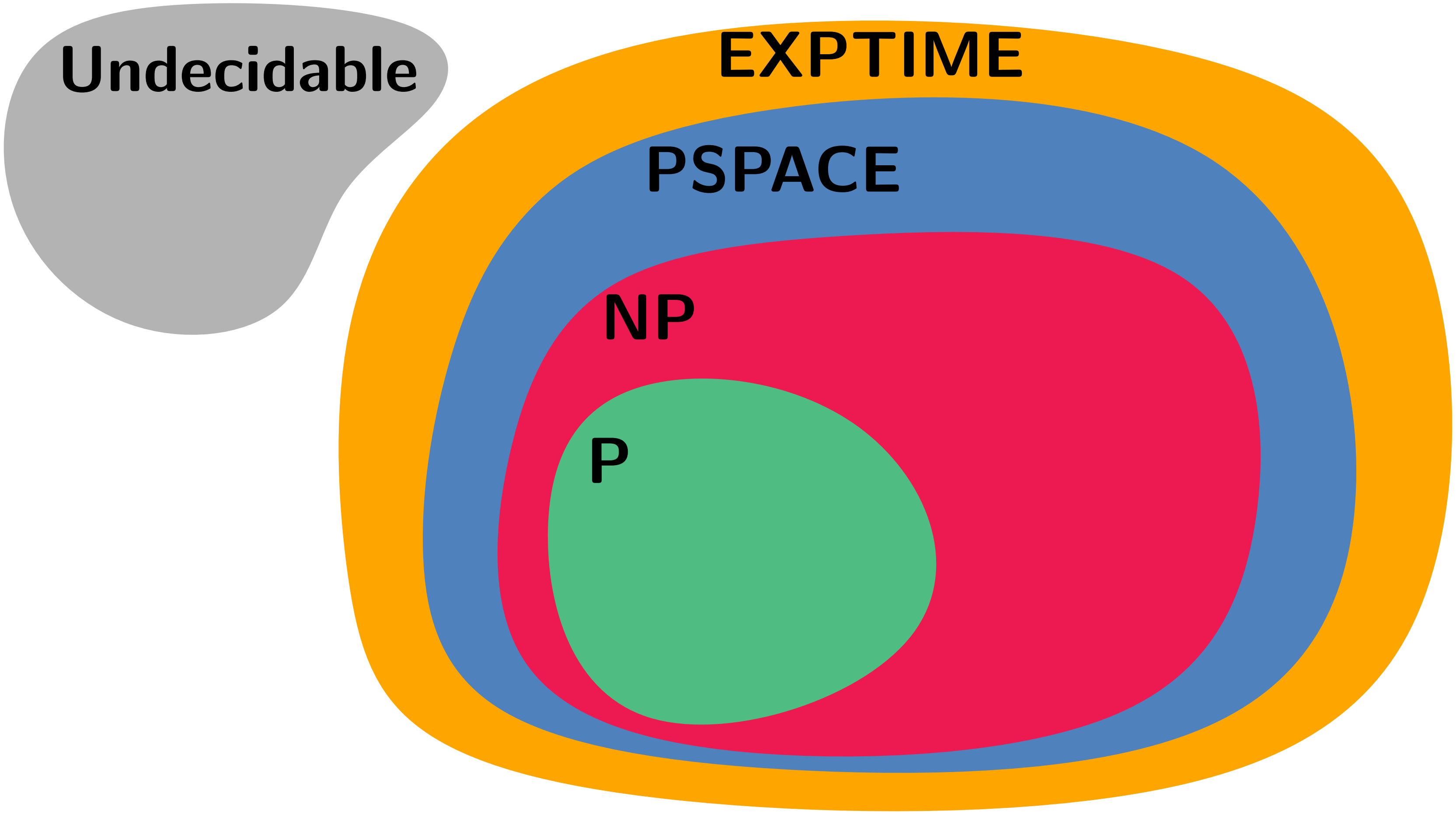
Some Complexity Classes



Question: Does **EXPTIME** include all decision problems?

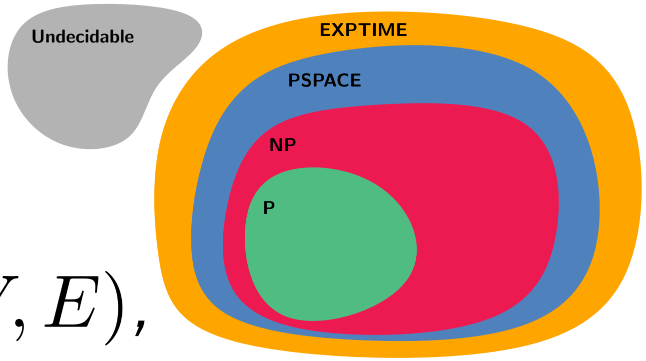
Answer: No, there are many more classes. One example:

Some Complexity Classes



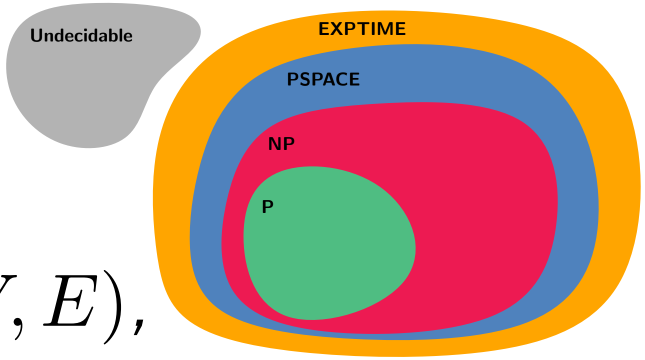
Undecidable problems are problems for which one can prove that there is no algorithm that always solves them, no matter how much time or space is allowed.

Some Example Problems



- Paths in graphs: *Given a graph $G = (V, E)$, is there a ...*
  - *path from vertex u to vertex v with at most k edges?*
  - *simple path from u to v with at least k edges?*
  - *simple path through all vertices (with $n-1$ edges)?*
- Integer factorization: *Given two integers n and k with $1 < k < n$, does n have a factor d with $1 < d \leq k$?*
- Halting problem: *Given a program P and an input I,*
  - *does P halt on I after finitely many steps?*
  - *does P halt on I after exponentially many steps?*
- Checkers/Hex: *Given an $n \times n$ board and a game situation, is there a winning strategy for the first player?*

Some Example Problems

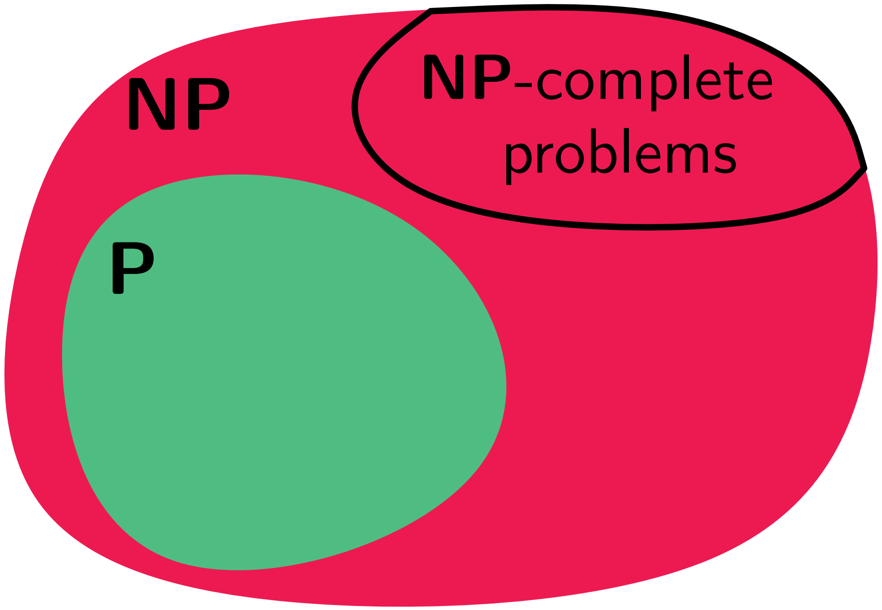


- Paths in graphs: *Given a graph $G = (V, E)$, is there a ...*
  - *path from vertex u to vertex v with at most k edges?*
  - *simple path from u to v with at least k edges?*
  - *simple path through all vertices (with $n - 1$ edges)?*
- Integer factorization: *Given two integers n and k with $1 < k < n$, does n have a factor d with $1 < d \leq k$?*
- Halting problem: *Given a program P and an input I,*
  - *does P halt on I after finitely many steps?*
  - *does P halt on I after exponentially many steps?*
- Checkers/Hex: *Given an $n \times n$ board and a game situation, is there a winning strategy for the first player?*

Some Complexity Classes

Next: Concentrate just on **P** and **NP** . . .

- The **"easiest"** problems in **NP** are the ones in **P**.



- The **"hardest"** problems in **NP** are called **NP-complete**.
- **Reductions** are a tool to compare two problems with respect to "how hard" they are.

Reductions

A problem A is “at most as hard” as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).

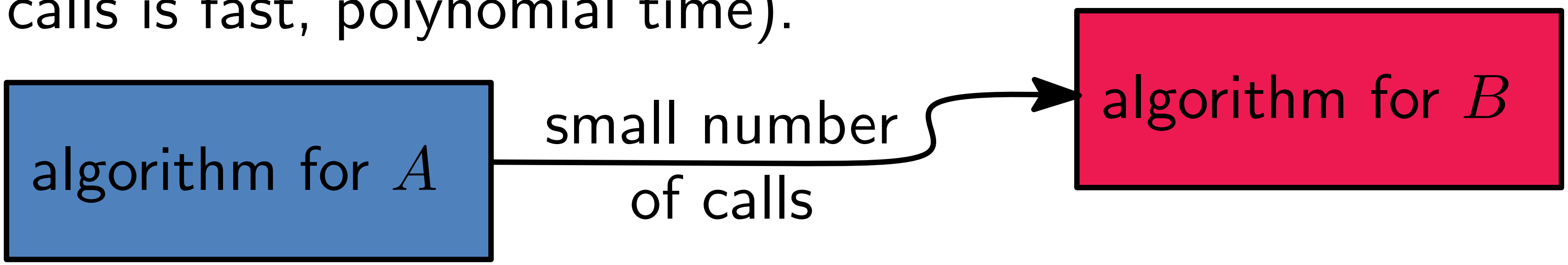
algorithm for A $\xrightarrow{\text{small number of calls}}$ algorithm for B

We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\mathbf{A} \leq_x \mathbf{B}}$

type of reduction

Reductions

A problem A is "at most as hard" as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).

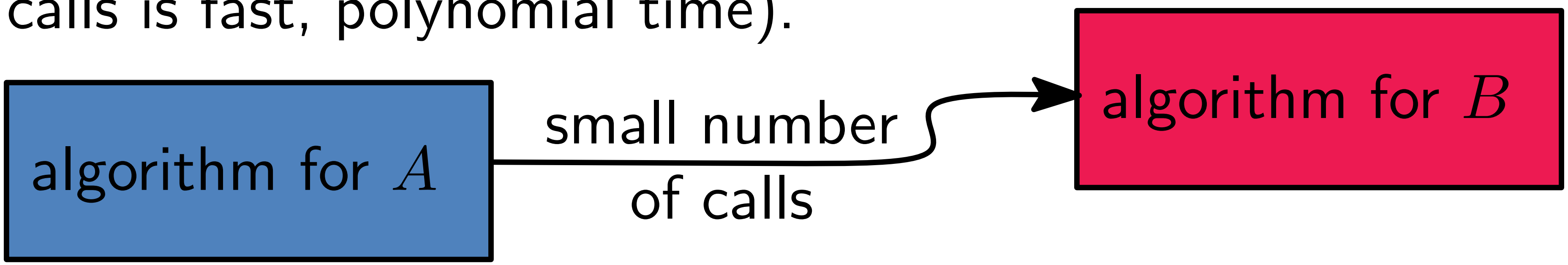


We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\mathbf{A} \leq_x \mathbf{B}}$

Note:
In a reduction from A to B, usually the "tricky" part of computing A is solved via B.

Reductions

A problem A is "at most as hard" as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).

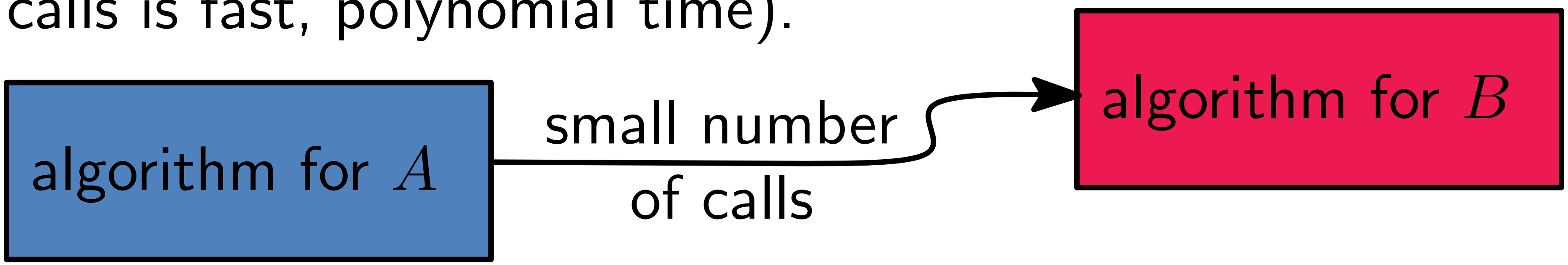


We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\textbf{A} \leq_x \textbf{B}}$

Note:
"*Reducing* A to B" means "*solving* A *with the help of* B".
(It does <u>not</u> mean "making A smaller to obtain B").

Reductions

A problem A is "at most as hard" as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).



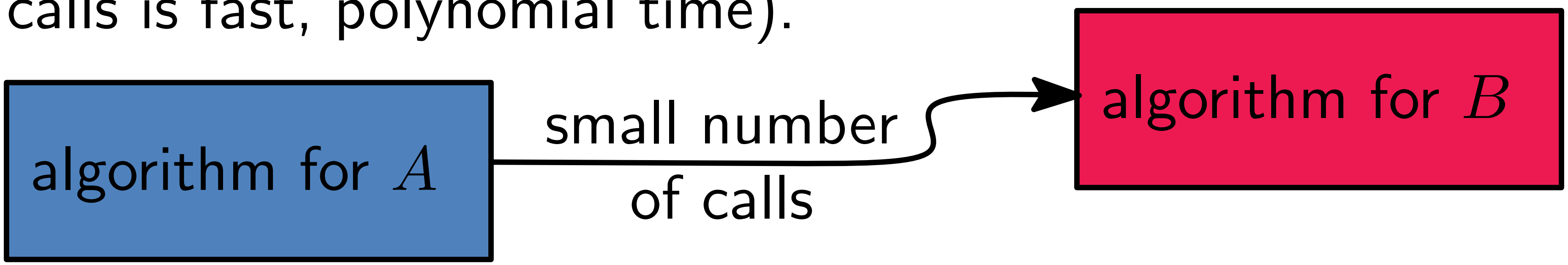
We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\mathbf{A} \leq_x \mathbf{B}}$

Polynomial time Turing (Cook) reduction A $\leq_{PT}$ B:

- At most polynomially many calls to the subroutine for B.
- Everything except the subroutine calls for B needs polynomial time in total.

Reductions

A problem A is "at most as hard" as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).

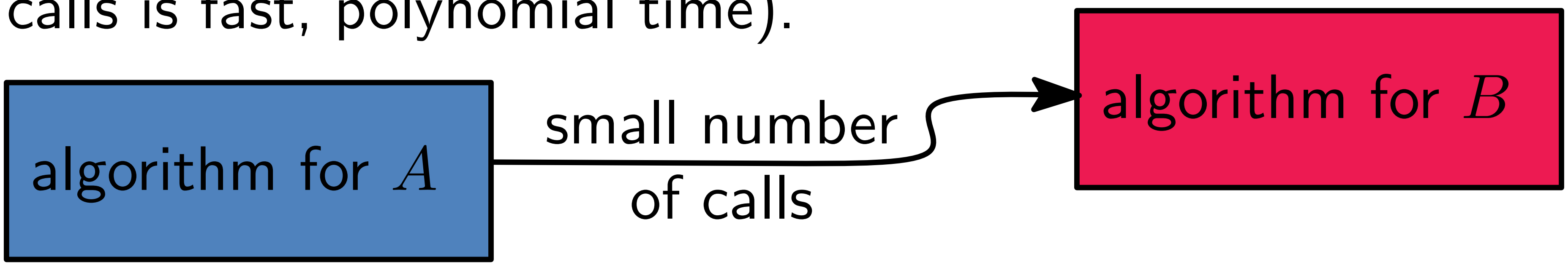


We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\mathbf{A} \leq_x \mathbf{B}}$

Polynomial time Karp reduction $\mathbf{A} \leq_p \mathbf{B}$:
Transform inputs for A into inputs for B in polynomial time, in a way that the output from B on the transformed input is the same as the output from A for the original input.

Reductions

A problem A is “at most as hard” as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).

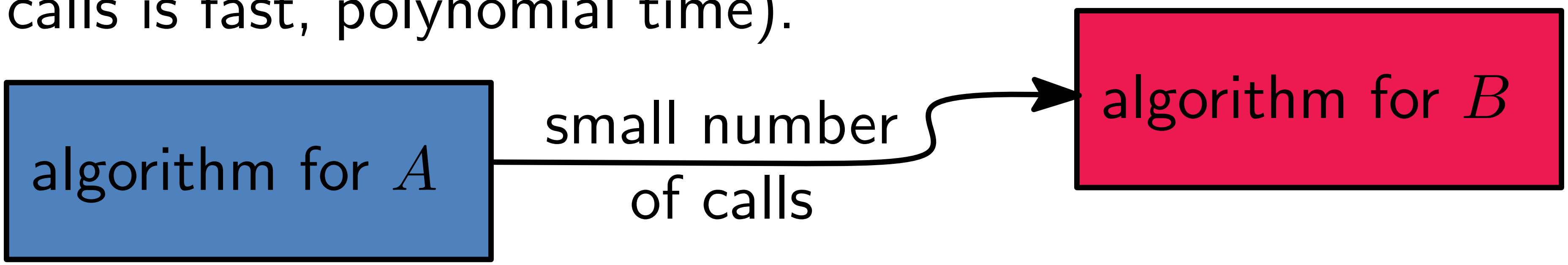


We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\mathbf{A} \leq_x \mathbf{B}}$

Note: Polynomial time Karp reductions are a special case of polynomial time Turing reductions.

Reductions

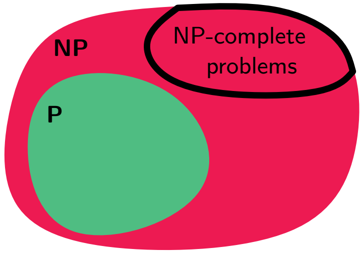
A problem A is "at most as hard" as a problem B if we can make an algorithm for solving A that uses a small number of calls to a subroutine for B (everything outside the subroutine calls is fast, polynomial time).



We say that $\boxed{\textbf{A is reduced to B}}$ and write $\boxed{\mathbf{A} \leq_x \mathbf{B}}$

Note: A $\leq_{PT}$ B or A $\leq_p$ B does not imply that an algorithm for A runs faster than one for B. But it implies that

- if B is in **P**, then A is in **P** as well.
- if A is not in **P** then B can't be in **P** either.



NP-completeness

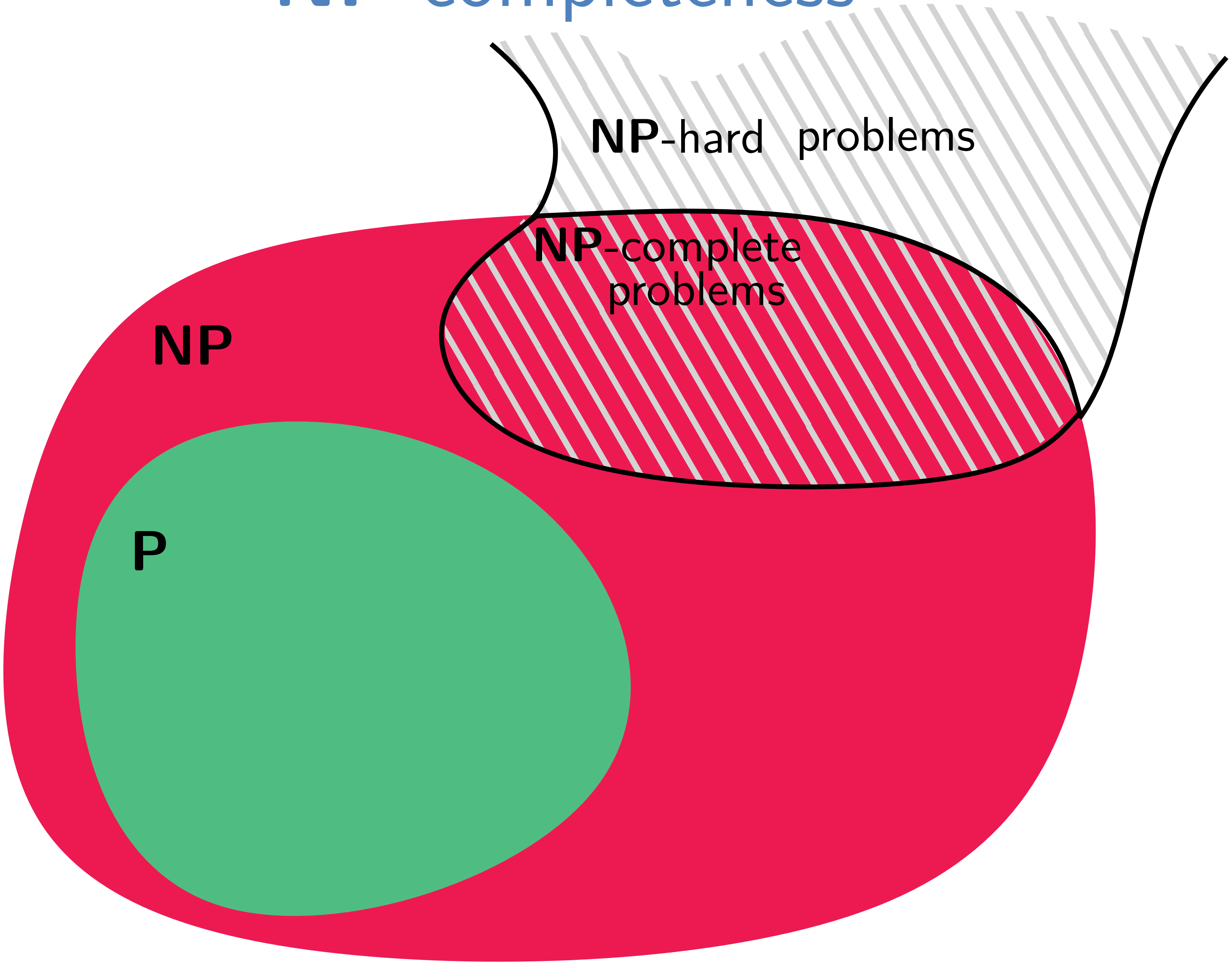
Now we are ready to formally define **NP**-completeness.

- A problem B is **NP-complete** if
  1. B is in **NP**, and
  2. B is "at least as hard" as all other problems in **NP**, or, more formally:
     A $\leq_p$ B for all problems A in NP.

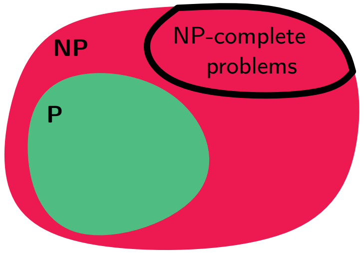
- A problem B is **NP-hard** if B is "at least as hard" as all problems in **NP**.

$\Rightarrow$ B is **NP-complete** if B is in **NP** and **NP-hard**.

NP-completeness



NP-completeness



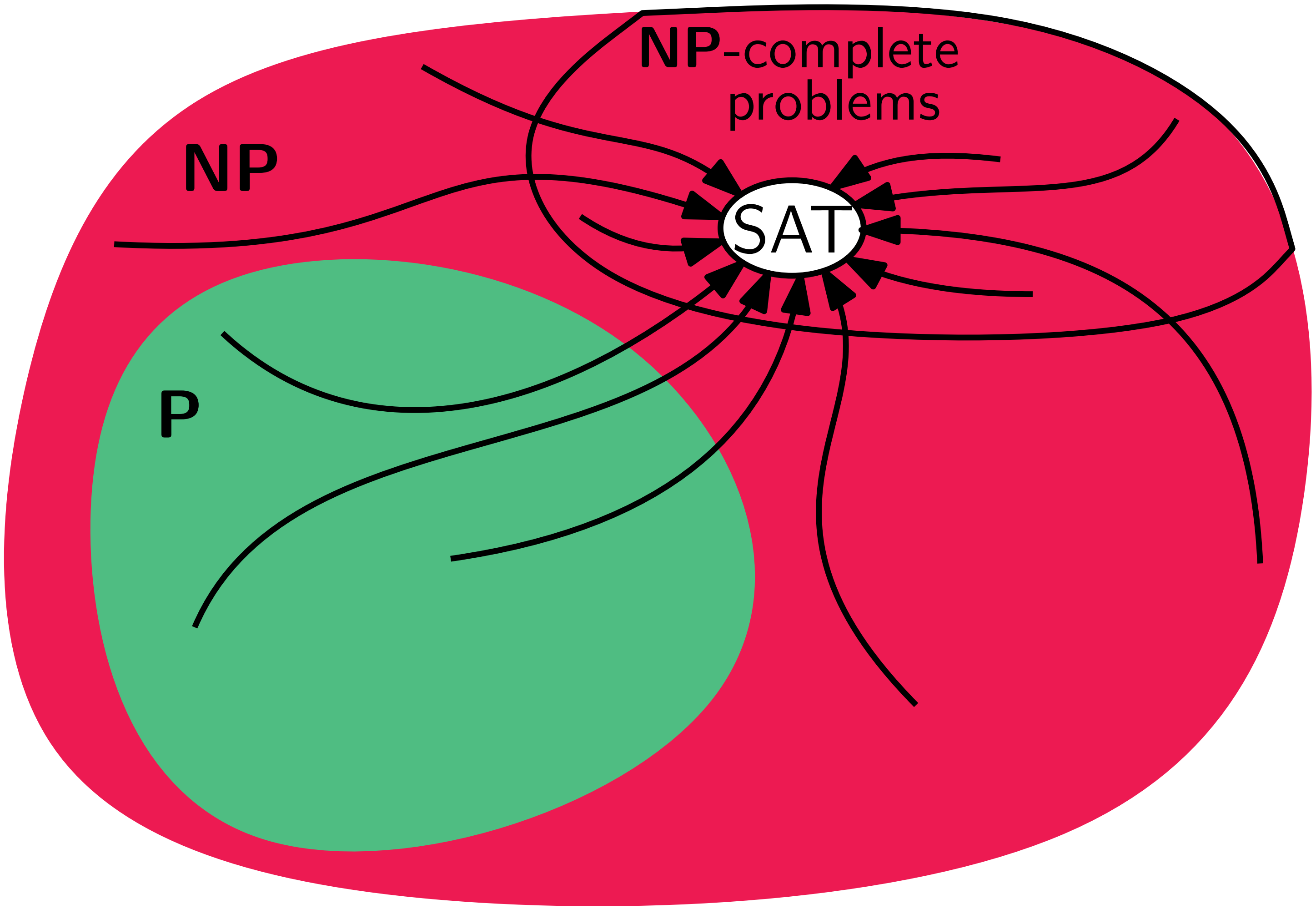
How can we show that a problem B in **NP** is **NP-complete**?

- Possibility 1:
  Show A $\leq_p$ B for **all problems** A **in NP**.

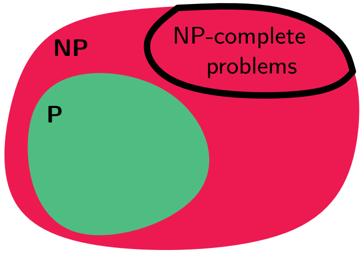
  Cook's Theorem:
  SAT (satisfiability of boolean formulas) is **NP-complete**.

NP-completeness

Cook's Theorem: A $\leq_p$ SAT for all A in **NP**



NP-completeness



How can we show that a problem B in **NP** is **NP-complete**?

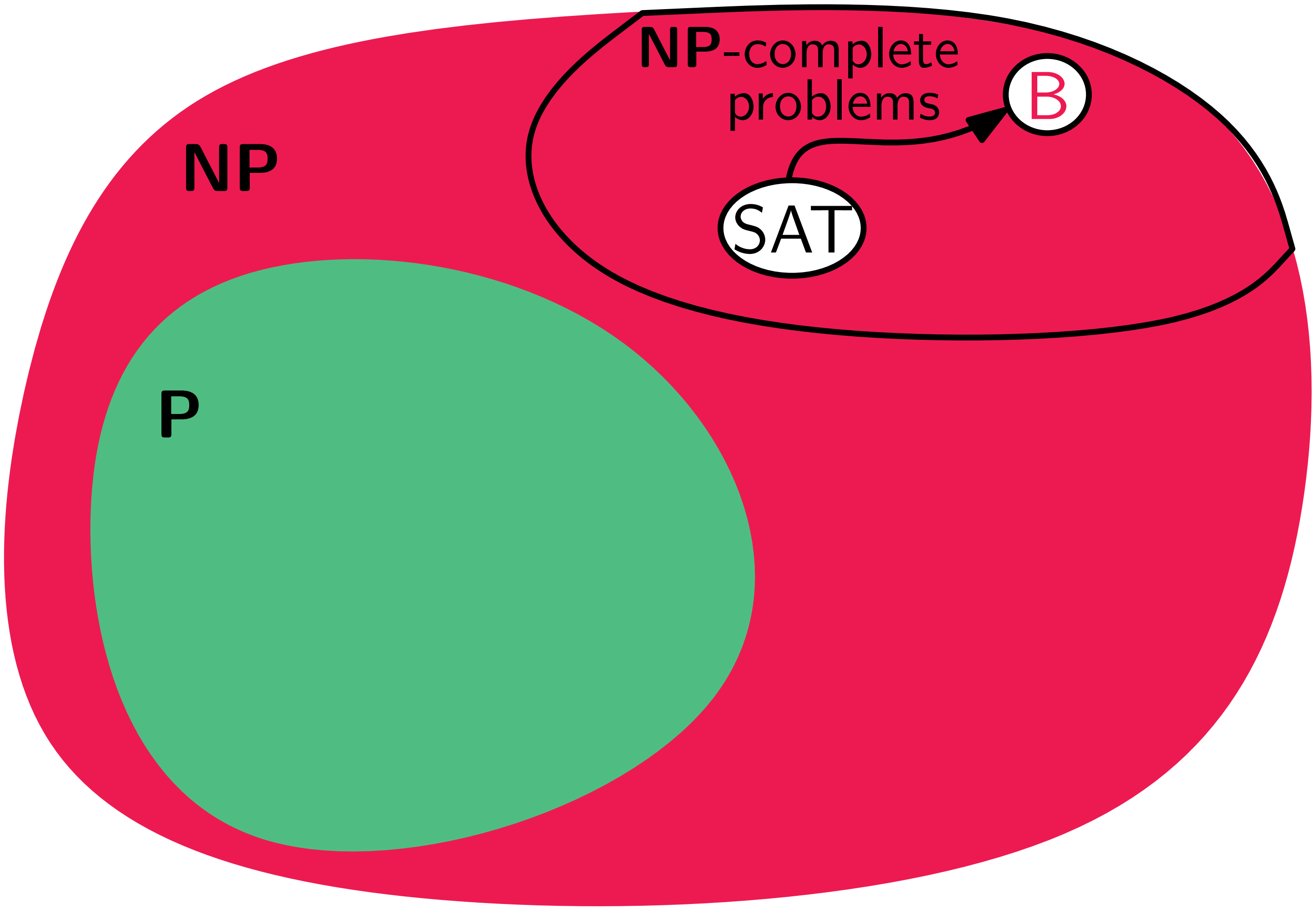
- Possibility 1:
  Show A $\leq_p$ B for **all problems** A **in NP**.

  Cook's Theorem:
  SAT (satisfiability of boolean formulas) is **NP-complete**.

- Possibility 2:
  Show C $\leq_p$ B for **some NP-complete problem** C:

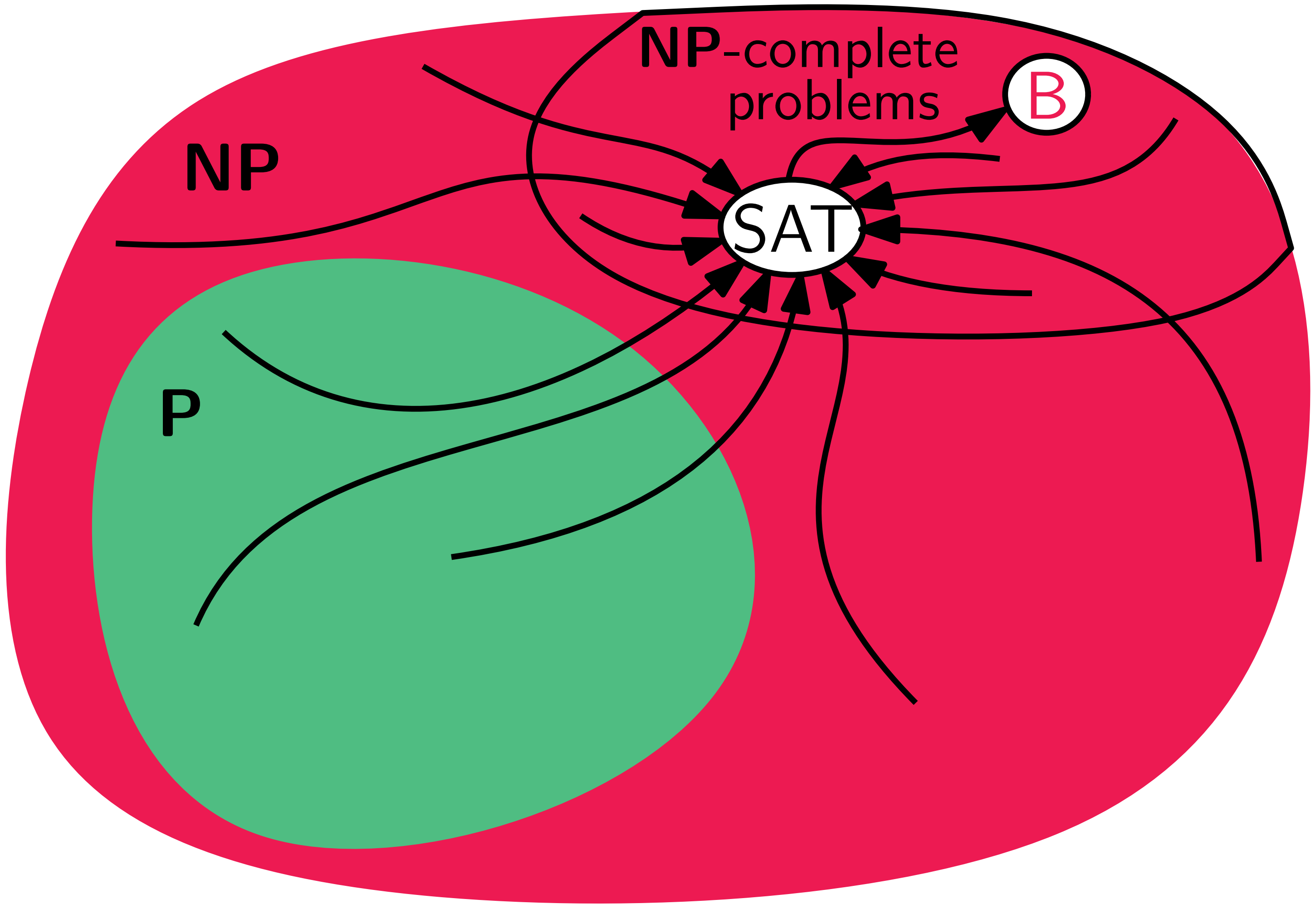
NP-completeness

Example: SAT $\leq_p$ B



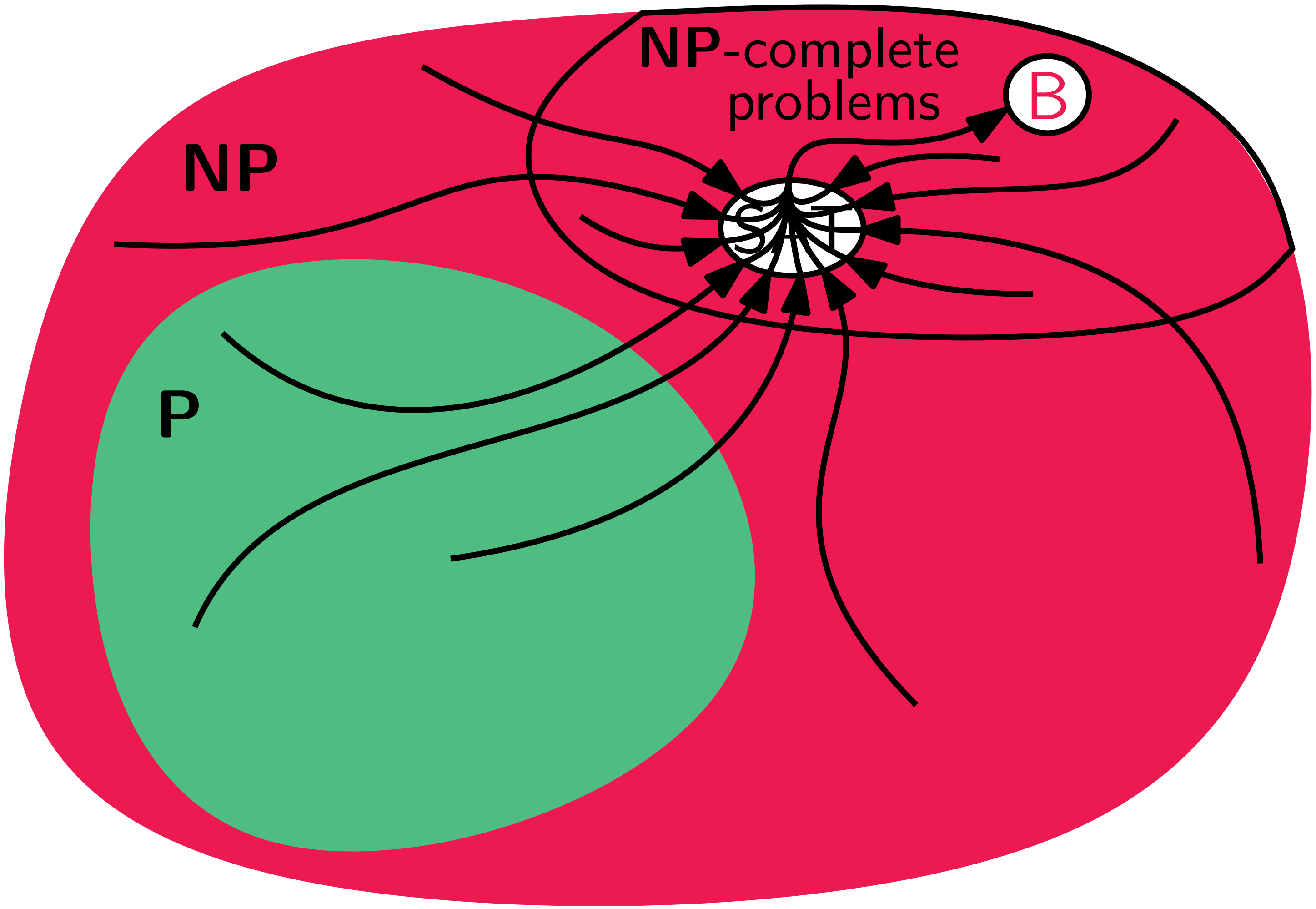
NP-completeness

Example: SAT $\leq_p$ B

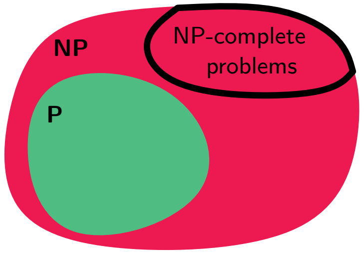


NP-completeness

Example: SAT $\leq_p$ B



NP-completeness



How can we show that a problem B in **NP** is **NP-complete**?

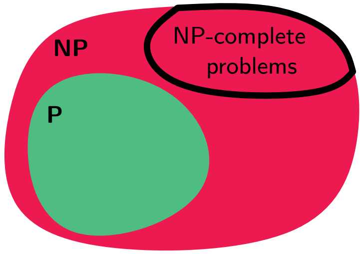
- Possibility 1:
  Show A $\leq_p$ B for **all problems** A **in NP**.

  Cook's Theorem:
  SAT (satisfiability of boolean formulas) is **NP-complete**.

- Possibility 2:
  Show C $\leq_p$ B for **some NP-complete problem** C:

  As A $\leq_p$ C for all problems A **in NP**,
  and as A $\leq_p$ C and C $\leq_p$ B implies A $\leq_p$ B,
  it follows that A $\leq_p$ B for all problems A **in NP**.

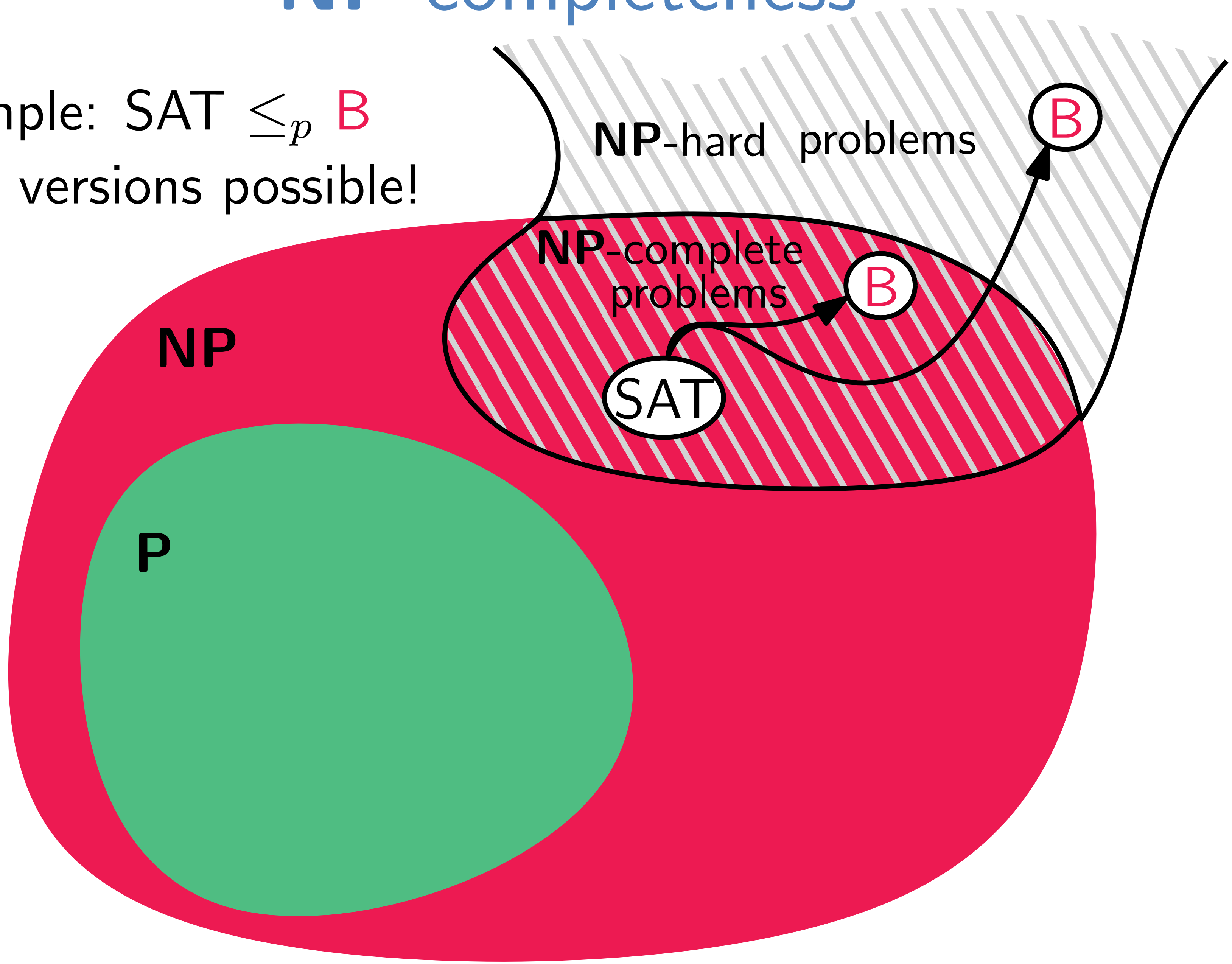
NP-completeness



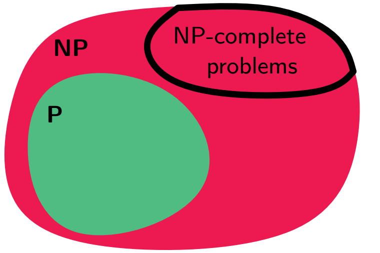
How can we show that a problem B is **NP-complete**?

NP-completeness

Example: SAT $\leq_p$ B
Both versions possible!



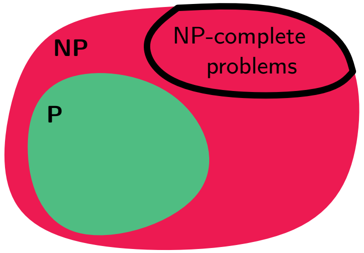
NP-completeness



How can we show that a problem B is **NP-complete**?

$\Rightarrow$ As before, but show additionally that B is in **NP**.

NP-completeness



How can we show that a problem B is **NP-complete**?

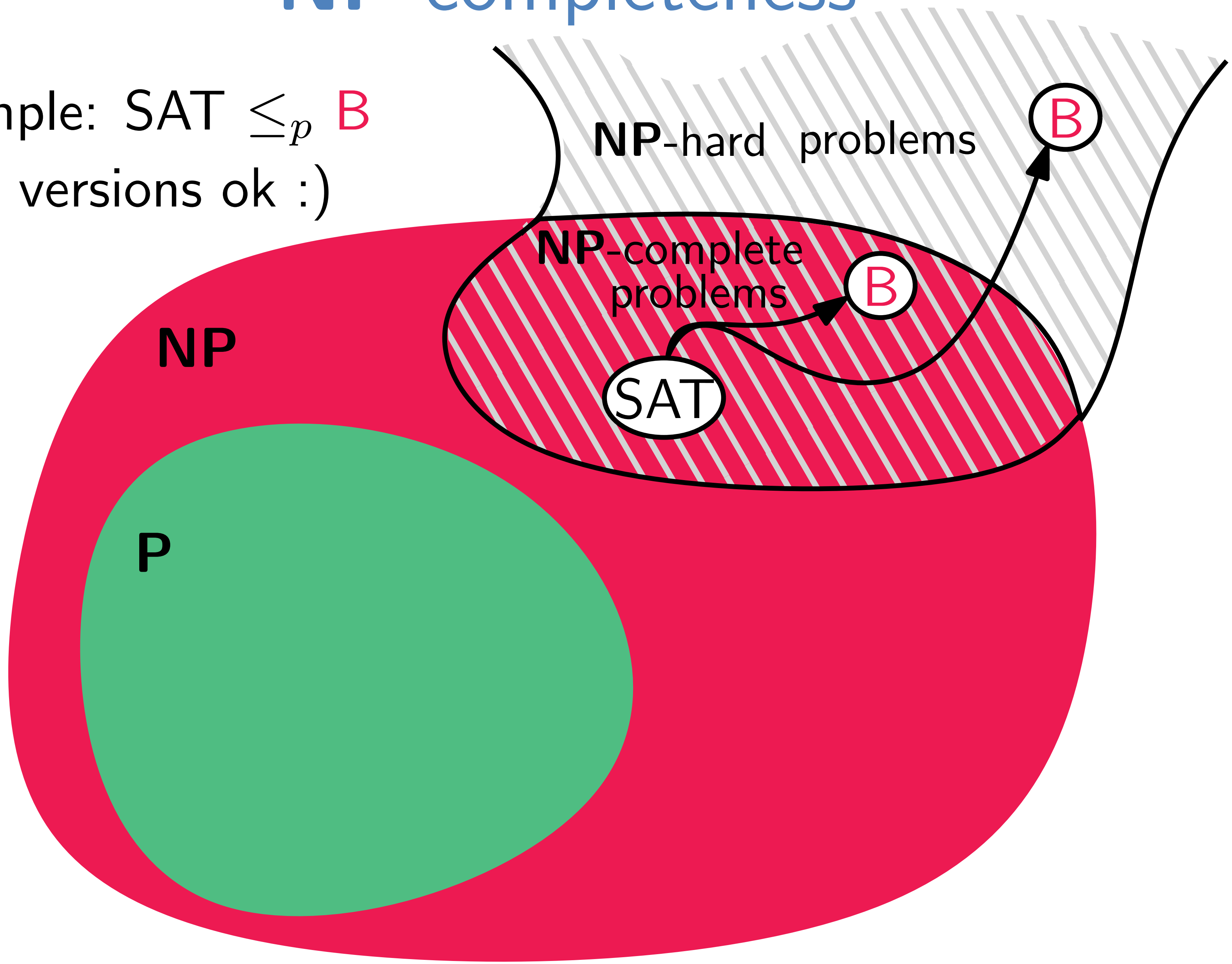
$\Rightarrow$ As before, but show additionally that B is in **NP**.

How can we show that a problem B is **NP-hard**?

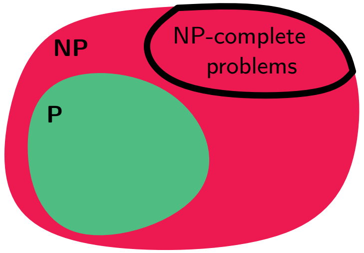
NP-completeness

Example: SAT $\leq_p$ B

Both versions ok :)



NP-completeness



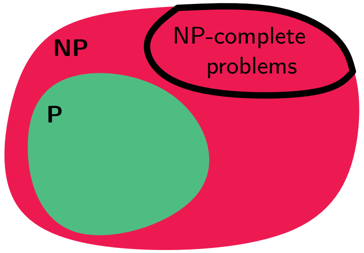
How can we show that a problem B is **NP-complete**?

$\Rightarrow$ As before, but show additionally that B is in **NP**.

How can we show that a problem B is **NP-hard**?

$\Rightarrow$ As before.

NP-completeness



Examples of **NP**-complete problems:

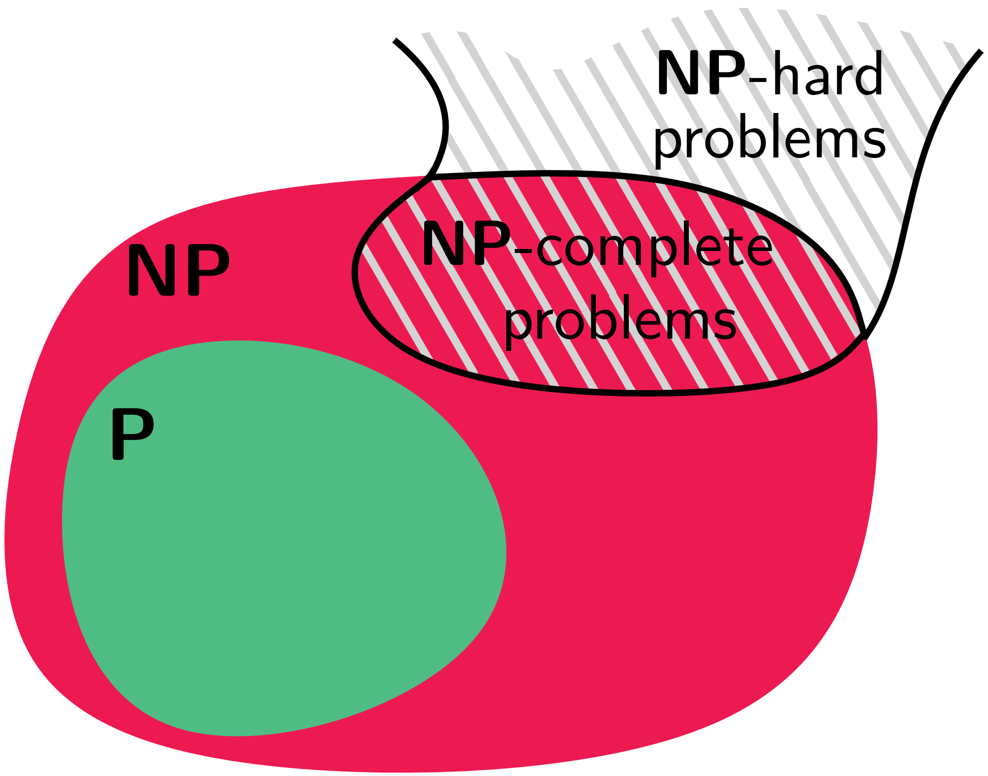
- SAT, 3SAT (Satisfiability of boolean formulas in 3CNF)
- Existence of Hamiltonian cycle, Hamiltonian path
- Longest path (decision version)
- TSP (Travelling Salesman Problem, decision version)
- Largest independent set, clique (decision version)
- Several graph coloring problems

... and many more! See for example the book
Computers and Intractibility:
A guide to the theory of NP-completeness.
by Michael R. Garey and David S. Johnson.

NP-completeness

Let's look once more at the picture ...

Question: Is it true that $\mathbf{P} \subsetneq \mathbf{NP}$, like shown here?



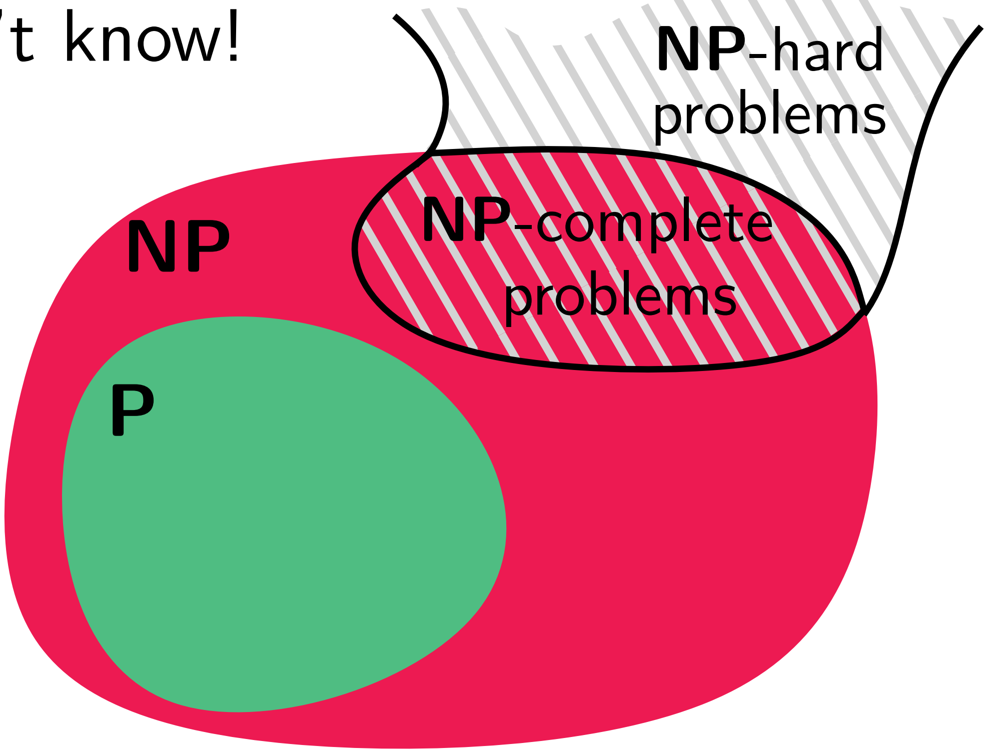
Stated differently: If it is always "easy" to verify a positive answer via a "proof", can it still be "hard" to find the solution?

NP-completeness

Let's look once more at the picture ...

Question: Is it true that **P** $\subsetneq$ **NP**, like shown here?

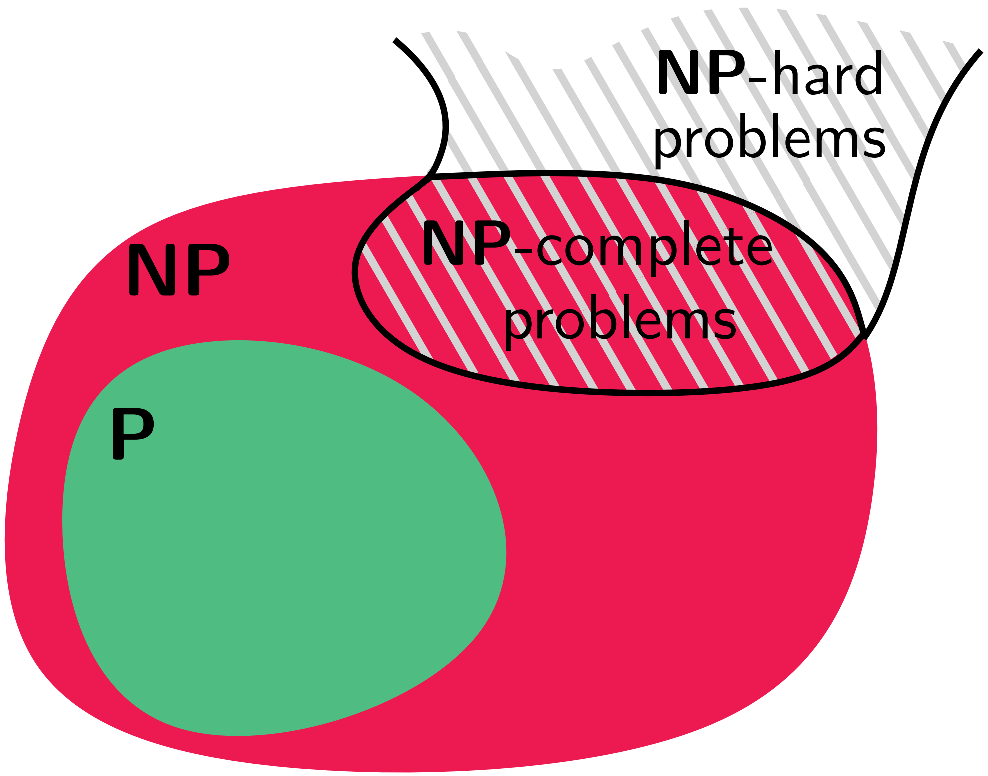
Answer: We don't know!



Stated differently: If it is always "easy" to verify a positive answer via a "proof", can it still be "hard" to find the solution?

The Question **P = NP ?**

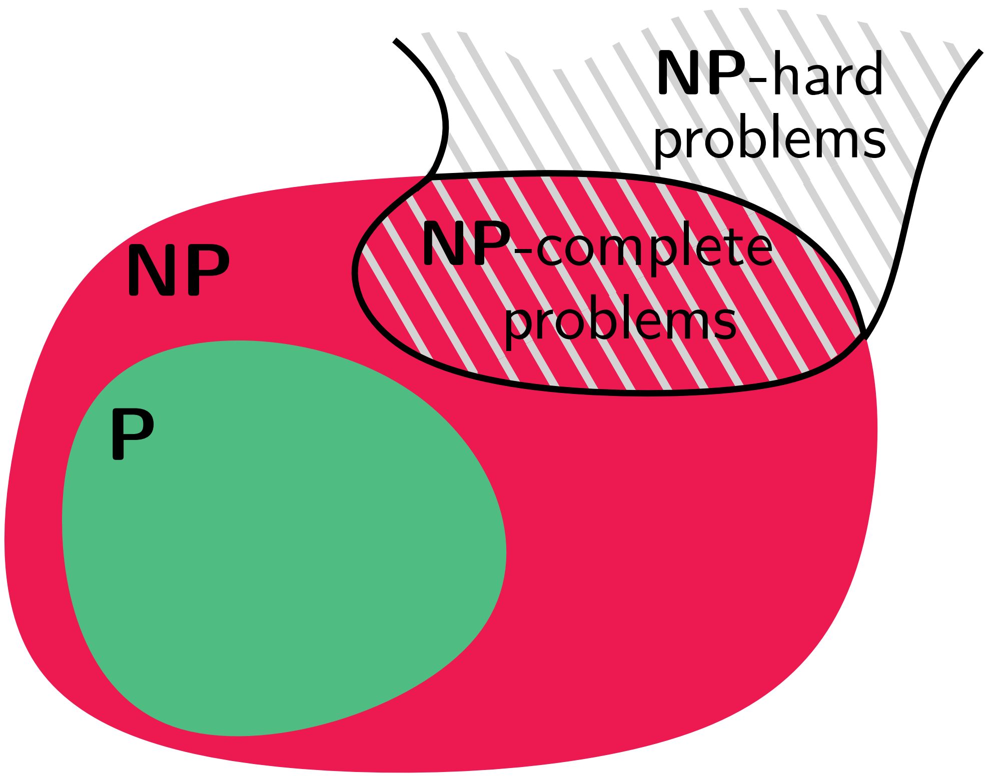
P = NP ? is the maybe most fundamental question in theoretical computer science.



Resolving it would bring great honor and also “fortune”: see **www.claymath.org/millennium-problems**

The Question **P = NP** ?

P = NP ? is the maybe most fundamental question in theoretical computer science.

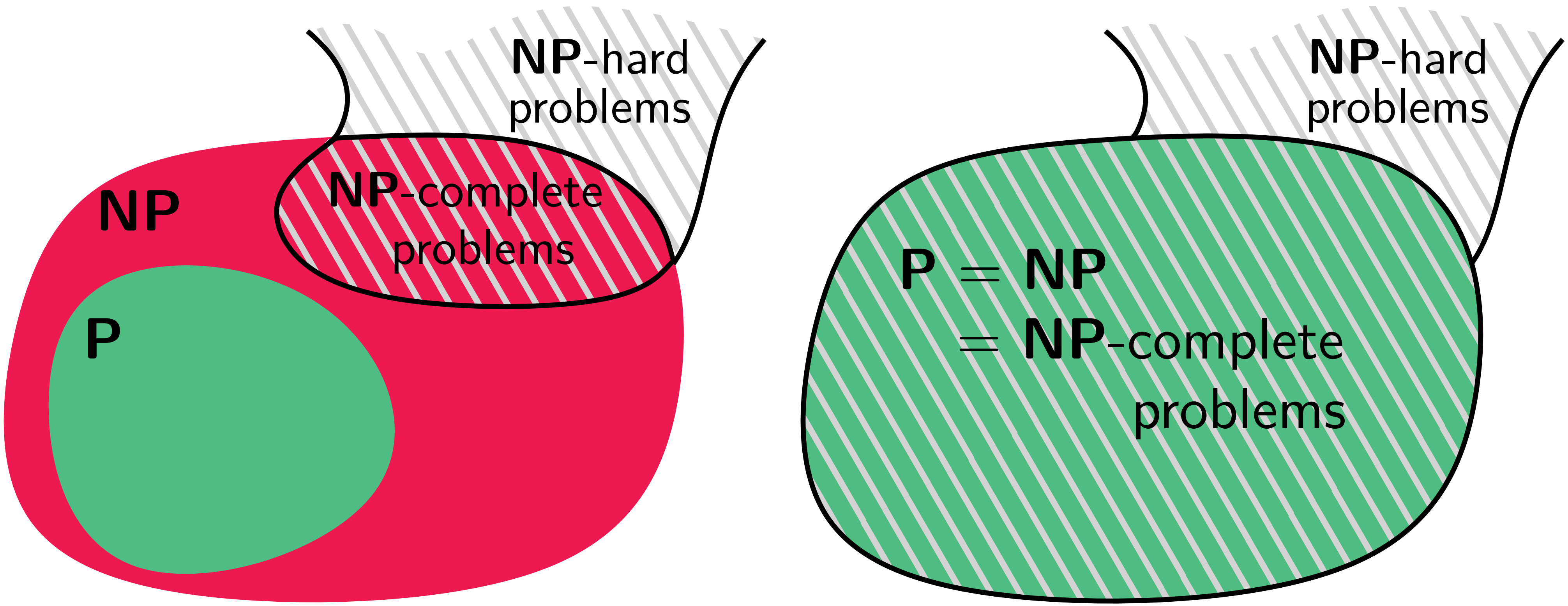


Question:

How would the picture in case of **P = NP** look like?

The Question P = NP ?

P = NP ? is the maybe most fundamental question in theoretical computer science.

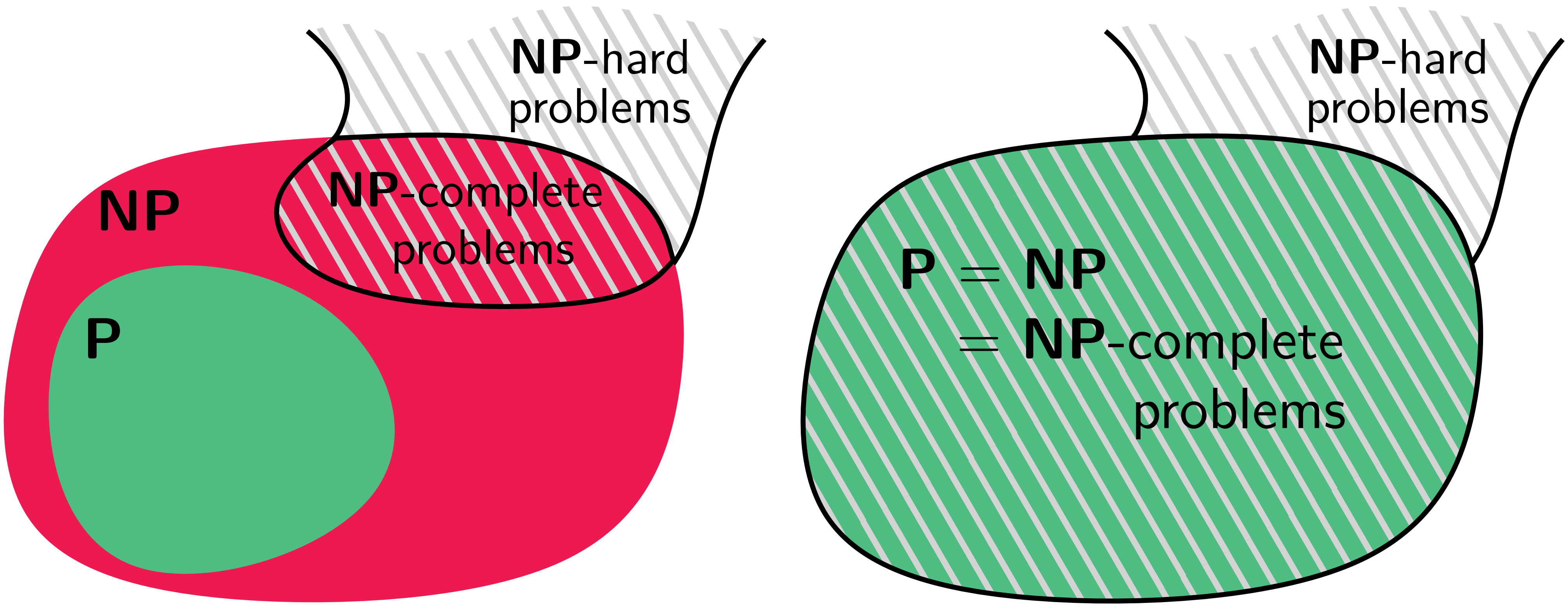


P $\neq$ **NP**

P = **NP**

The Question $\mathbf{P} = \mathbf{NP}$?

$\mathbf{P} = \mathbf{NP}$? is the maybe most fundamental question in theoretical computer science.



Question: How could one prove $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$?

Summary

- Two introduction problems:
  - Dinner party (Hamiltonian cycle)
  - City Tour (Hamiltonian path)
  - Sketch of a reduction: City Tour $\leq_p$ Dinner Party
- Some complexity classes: **P, NP, PSPACE, EXP**
- Polynomial time reductions: Turing ($\leq_{PT}$), Karp ($\leq_p$)
- **NP**-hardness and **NP**-completeness:
  Definition, example problems, the question **P = NP ?**
- Questions: Discussion session

Thank you for your attention.