CS 344: Design and Analysis of Computer Algorithms

(Spring 2022 — Sections 5,6,7,8)

Lecture 23:
Introduction to P vs NP

This week's topics:

- Efficient Algorithms: Polynomial Runtime
- Decision Problems
- Solving vs Verifying a Decision Problem
- Complexity Classes P & NP
- NP-Hard & NP-Complete problems
- Circuit-SAT Problem & Cook-Levin Theorem

Efficient Algorithms: Polynomial Runtime

Efficient Algorithm

- For an algorithm to be practical, it should be "efficient”
- Its runtime should not be too large
- A minimal requirement for efficiency of runtime:
  - The algorithm should run in polynomial time
  - The runtime should be $O(N^k)$ when N is the input size and k is any arbitrary constant

Efficient Algorithm

- For an algorithm to be practical, it should be "efficient”
- Its runtime should not be too large
- A minimal requirement for efficiency of runtime:
  - The algorithm should run in polynomial time
  - The runtime should be $O(N^k)$ when N is the input size and k is any arbitrary constant
- Polynomial time: $O(N), \quad O(N^2), \quad O(N^{100}), \quad O(\log(N)), \quad O(\sqrt{N})$
- NOT polynomial time: $O(2^N), \quad O(N!), \quad O(N^N), \quad O(2^{2^N})$

Polynomial Time Algorithm

- Almost all the algorithms we saw in this course were polynomial time algorithms:
  - Sorting: given n numbers, the algorithms sort them in $O(n^2)$ time

Polynomial Time Algorithm

- Almost all the algorithms we saw in this course were polynomial time algorithms:
  - Sorting: given n numbers, the algorithms sort them in $O(n^2)$ time

Input size N $\approx n$

Runtime $\approx O(N^2)$

Polynomial Time Algorithm

- Almost all the algorithms we saw in this course were polynomial time algorithms:
  - Sorting: given n numbers, the algorithms sort them in $O(n^2)$ time
  - Dynamic programming, say, for LIS: given n numbers, the algorithms find the length of LIS in $O(n^2)$ time

Polynomial Time Algorithm

- Almost all the algorithms we saw in this course were polynomial time algorithms:
  - Sorting: given n numbers, the algorithms sort them in $O(n^2)$ time
  - Dynamic programming, say, for LIS: given n numbers, the algorithms find the length of LIS in $O(n^2)$ time

Input size N $\approx n$

Runtime $\approx O(N^2)$

Polynomial Time Algorithm

- Almost all the algorithms we saw in this course were polynomial time algorithms:
  - Sorting: given n numbers, the algorithms sort them in $O(n^2)$ time
  - Dynamic programming, say, for LIS: given n numbers, the algorithms find the length of LIS in $O(n^2)$ time
  - Graph algorithms, say, Shortest Path: given a graph with n vertices and m edges, the Bellman-Ford algorithm finds the shortest path in $O(m \cdot n)$ time

Polynomial Time Algorithm

- Almost all the algorithms we saw in this course were polynomial time algorithms:
  - Sorting: given n numbers, the algorithms sort them in $O(n^2)$ time
  - Dynamic programming, say, for LIS: given n numbers, the algorithms find the length of LIS in $O(n^2)$ time
  - Graph algorithms, say, Shortest Path: given a graph with n vertices and m edges, the Bellman-Ford algorithm finds the shortest path in $O(m \cdot n)$ time

Input size N $\approx n + m = O(n^2)$

Runtime $\approx O(N^{3/2})$

Polynomial Time Algorithm

- **Tricky Question:**
  - Did the dynamic programming algorithm for Fibonacci number run in polynomial time?
    - Input: the number n itself
    - Runtime: $O(n)$ time

Polynomial Time Algorithm

- **Tricky Question:**
  - Did the dynamic programming algorithm for Fibonacci number run in polynomial time?
    - Input: the number n itself
    - Runtime: O(n) time
  - No it is NOT

Polynomial Time Algorithm

- **Tricky Question:**
  - Did the dynamic programming algorithm for Fibonacci number run in polynomial time?
    - Input: the number n itself
    - Runtime: O(n) time
  - No it is NOT
  - The input size is NOT n here, it is just a single number
  - So $N = O(\log n)$ and the runtime is $O(n) = O(2^N)$

NOT Poly-time Algorithms

- ALL of this course has been about designing efficient algorithms:
  - How can we show a problem P can be solved in poly-time?

NOT Poly-time Algorithms

- ALL of this course has been about designing efficient algorithms:
  - How can we show a problem P can be solved in poly-time?
- What if we instead want to show that a problem P **cannot** be solved in poly-time?

NOT Poly-time Algorithms

- ALL of this course has been about designing efficient algorithms:
  - How can we show a problem P can be solved in poly-time?
- What if we instead want to show that a problem P **cannot** be solved in poly-time?
- Why do we want to do that?

NOT Poly-time Algorithms

- ALL of this course has been about designing efficient algorithms:
  - How can we show a problem P can be solved in poly-time?
- What if we instead want to show that a problem P **cannot** be solved in poly-time?
- Why do we want to do that?
  - So we do not waste our time trying to design an efficient algorithm for P
  - So we can use P to design passwords, do cryptography, or create bitcoins,…

Decision Problems

Decision Problem

- **Problem**: a mapping from inputs to valid outputs
- **Decision Problem**: when the output is either YES or NO only

Decision Problem

- **Problem**: a mapping from inputs to valid outputs
- **Decision Problem**: when the output is either YES or NO only
- Examples of decision problems:
  - Given an array A and number x, does x appear in A?
  - Given a knapsack of size W and n items with different sizes, is there a way to make the knapsack completely full?
  - Given a graph G, is G connected?
  - Given a directed graph G, is there a cycle in G?

Decision Problem

- **Problem**: a mapping from inputs to valid outputs
- **Decision Problem**: when the output is either YES or NO only
- Examples of **NOT** decision problems:
  - Given an array A, sort the array
  - Given a graph G, find an MST of G
  - Given a directed graph G and vertices s and t, find the shortest path from s to t

Decision Problem

- **Problem**: a mapping from inputs to valid outputs
- **Decision Problem**: when the output is either YES or NO only
- NOT Decision Problems:

Many problems that are NOT a decision problem, have many similar decision variants

  - Given a directed graph G and vertices s and t, find the shortest path from s to t

Decision Problem

- **Problem**: a mapping from inputs to valid outputs
- **Decision Problem**: when the output is either YES or NO only
- Some decision variants:
  - Given an array A, is the array A sorted?
  - Given a graph G and integer T, is the weight of MST of G at most T?
  - Given a directed graph G, vertices s and t, and integer T, is the weight of the shortest path from s to t at most T?

Solving vs Verifying a Decision Problem

Solving a Problem

- All of this course so far was about designing algorithms
- Algorithms solve a problem:
  - Given the input, algorithms find a correct output

Solving a Problem

- All of this course so far was about designing algorithms
- Algorithms solve a problem:
  - Given the input, algorithms find a correct output

- Example:
  - Decision problem: Given a graph G, is G connected?

Solving a Problem

- All of this course so far was about designing algorithms
- Algorithms solve a problem:
  - Given the input, algorithms find a correct output
- Example:
  - Decision problem: Given a graph G, is G connected?
  - DFS or BFS algorithms solve this problem

Verifying a Decision Problem

- Verifiers have a much simpler task than algorithms
- Consider the following scenario:
  - We have an input x to some decision problem P
  - I claim that the right answer is P(x) = YES
  - You then will ask me can you prove it though? If so provide me with your proof y
  - So I give you a proof y also
  - Can you verify, given the input x and the proof y, that P(x) = YES indeed?

Example 1

- **Decision problem:**
- Is the maximum flow in network G from s to t at least T?

Example 1

- **Decision problem:**
- Is the maximum flow in network G from s to t at least T?
- I claim YES.
- A natural proof you can ask for me is then to write down the flow function $f: V \times V \to \mathbb{R}$ as a proof
- If I give you the proof f also, can you convince yourself the answer to the problem is YES?
  - Just check f is a feasible flow (preservation + capacity)
  - If the value of f is at least T

Example 1

- **Decision problem:**
- Is the maximum flow in network G from s to t at least T?
- I claim YES

The algorithm you use to verify the correctness is called a **Verifier**

the problem is YES?

  - Just check f is a feasible flow (preservation + capacity)
  - If the value of f is at least T

Verifier

- A verifier for a decision problem P with input x:
- The verifier specifies what type of a proof y it needs
  - The burden of finding the proof is NOT on the verifier
  - The only requirement is that:
    - If P(x) = YES, a valid proof y should always exist
    - If P(x) = NO, there is no valid proof
- Given x and y, the verifier should output if P(x) = YES or not (alternatively, verify if the “proof” is correct or not)

Verifier

- **Runtime of verifier:**

- The time it takes, given x and y, for the verifier to decide if P(x) = YES or not

- The runtime is measured with respect to the size of the original input which is only x

- This is to prevent the verifier for asking a very long proof (otherwise, reading the proof itself takes too long)

Example 2

- **Decision problem:**
- Is the shortest path in graph G from s to t at most T?

Example 2

- **Decision problem:**
- Is the shortest path in graph G from s to t at most T?
- Ask for a path P of weight at most T from s to t

Example 2

- **Decision problem:**
- Is the shortest path in graph G from s to t at most T?
- Ask for a path P of weight at most T from s to t
- If the answer is YES, there is a proof and otherwise there is none
- Check if P is a valid path or not
  - all edges belong to G and it starts from s and ends in t
- Check if P has weight at most T or not

Example 2

- **Decision problem:**
- Is the shortest path in graph G from s to t at most T?
- Ask for a path P of weight at most T from s to t
- If the answer is YES, there is a proof and otherwise there is none
- Check if P is a valid path or not
  - all edges belong to G and it starts from s and ends in t
- Check if P has weight at most T or not
- **Runtime:** the path has at most n-1 edges; so it only takes O(n) time to run the verifier (if G is given in adjacency matrix format)

Example 2

- **Decision problem:**
- Is the shortest path in graph G from s to t at most T?
- There is a very simple verifier with O(n) time for this problem
- But if we want to solve this problem, we need to run Dijkstra which takes $O(n + m \log m)$ time (and it is not that easy)

Example 3

- **Decision problem:**
- Is the longest path in graph G from s to t at least T?

Example 3

- **Decision problem:**
- Is the longest path in graph G from s to t at least T?
- Ask for a path of weight at least T from s to t as the proof
- If the answer is YES, there is a proof and otherwise there is none
- Check if P is a valid path or not
  - all edges belong to G and it starts from s and ends in t
- Check if P has weight at least T or not

Example 3

- **Decision problem:**
- Is the longest path in graph G from s to t at least T?
- Ask for a path of weight at least T from s to t as the proof
- If the answer is YES, there is a proof and otherwise there is none
- Check if P is a valid path or not
  - all edges belong to G and it starts from s and ends in t
- Check if P has weight at least T or not
- **Runtime:** the path has at most n-1 edges; so it only takes O(n) time to run the verifier (if G is given in adjacency matrix format)

Example 3

- **Decision problem:**
- Is the longest path in graph G from s to t at least T?
- There is a very simple verifier with O(n) time for this problem
- But if we want to solve this problem, we (the entire people on earth) do NOT know an efficient algorithm with poly(n) runtime

(Complexity) Classes
P & NP

Classes P and NP

- We use class to refer to a collection of problems

Classes P and NP

- We use class to refer to a collection of problems
- Class P:
  - ALL problems that can be solved in polynomial time
- Class NP:
  - ALL problems that can be verified in polynomial time

Classes P an

Problems we can solve efficiently

- We use class to refer to a collection of p ems
- Class P:
  - ALL problems that can be solved in polynomial time
- Class NP:
  - ALL problems that can be verified in polynomial time

Problems we “hope” to be able to solve efficiently

C P

Polynomial time

- We use class to refer to a collection of problems
- Class P:
  - ALL problems that can be solved in polynomial time
- Class NP:
  - ALL problems that can be verified in polynomial time

Non-deterministic polynomial time

Classes P and NP

- Clearly any problem in P is also in NP
  - If we can solve a problem in poly-time, we can definitely verify it in poly-time

Classes P and NP

- Clearly any problem in P is also in NP
  - If we can solve a problem in poly-time, we can definitely verify it in poly-time
- Big open question of Computer Science:

Is P=NP or not?

Classes P and NP

- Most researchers believe that $P \neq NP$ but we are nowhere close proving (even much weaker versions of) this

- Proving $P \neq NP$ is somewhat opposite of what we do in this course:

  - Instead of giving an efficient algorithm for a problem (what we did in this course), we want to show there is NO efficient algorithm for a problem

NP-Hard & NP-Complete problems

Plan

- We have a problem Q in NP
- We want to show that Q is impossible to solve in polynomial time
- We do not know how to do that

Plan

- We have a problem Q in NP
- We want to show that Q is impossible to solve in polynomial time
- We do not know how to do that
- Instead, we go for the next best thing:
  - Show that Q is really hard to solve in polynomial time

Plan

- We have a problem Q in NP
- We want to show that Q is impossible to solve in polynomial time
- We do not know how to do that
- Instead, we go for the next best thing:
  - Show that Q is really hard to solve in polynomial time
- **A simple approach:**
  - Show that if Q can be solved in polynomial time, then P = NP

Plan

- **A simple approach:**
  - Show that if Q can be solved in polynomial time, then P = NP
- How can we show such a thing?

Plan

- **A simple approach:**
  - Show that if Q can be solved in polynomial time, then P = NP
- How can we show such a thing? **Reductions!**
- Show that:
  - if there is a poly-time algorithm A for problem Q, then
  - we can use A in a black-box way to design a poly-time algorithm for every problem in NP

Plan

- **A simple approach:**
  - Show that if Q can be solved in polynomial time, then P = NP
- How can we show such a thing? Reductions!
- Show that:
  - if there is a poly-time algorithm A for problem Q, then
  - we can use A in a black-box way to design a poly-time algorithm for every problem in NP
- **Still NOT an easy plan:** we have to design infinitely many algorithms from A for every possible problem in NP

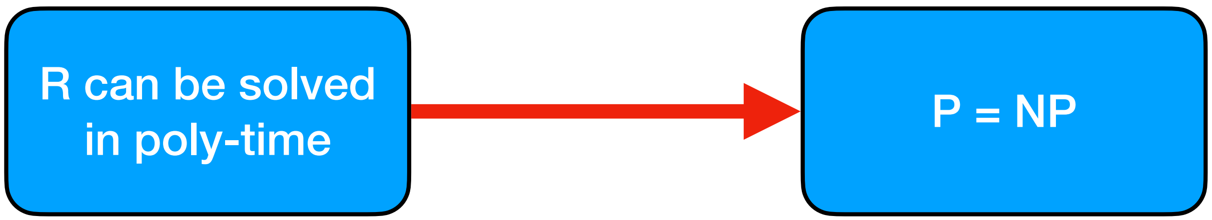
Plan

- **Still NOT an easy plan:** we have to design infinitely many algorithms from A for every possible problem in NP
- What if I tell you there is a single problem R such that if R can be solved in poly-time, then P=NP?

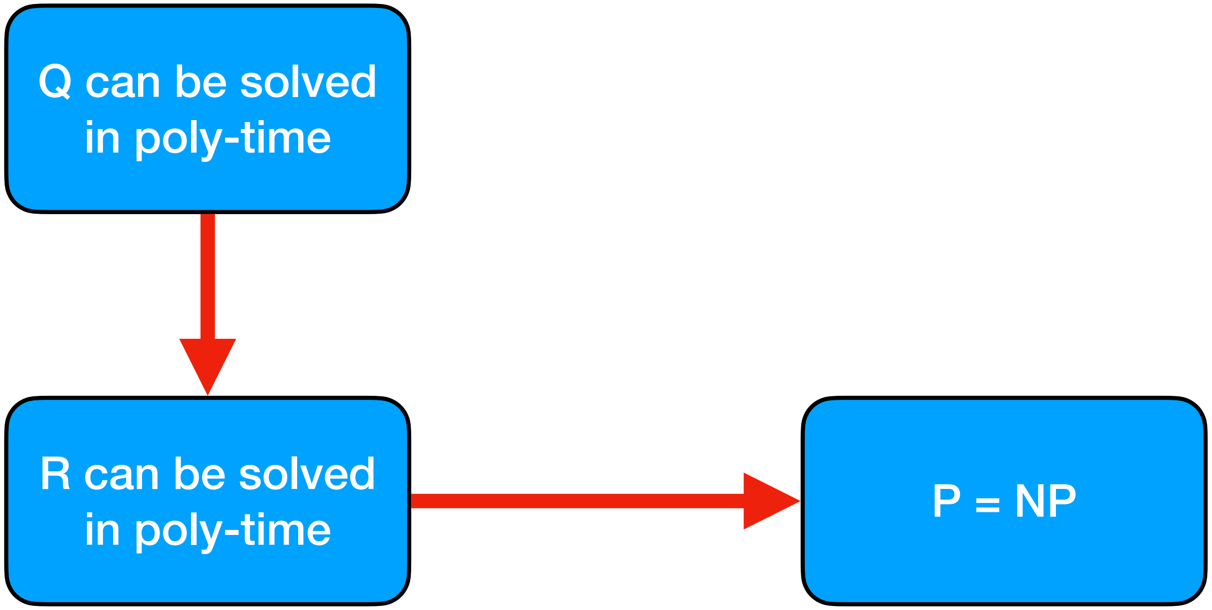
Plan

- **Still NOT an easy plan:** we have to design infinitely many algorithms from A for every possible problem in NP
- What if I tell you there is a single problem R such that if R can be solved in poly-time, then P=NP?
- Then we only need to do a reduction from R:
  - Show that the poly-time algorithm A for Q can be used to design a poly-time algorithm for R

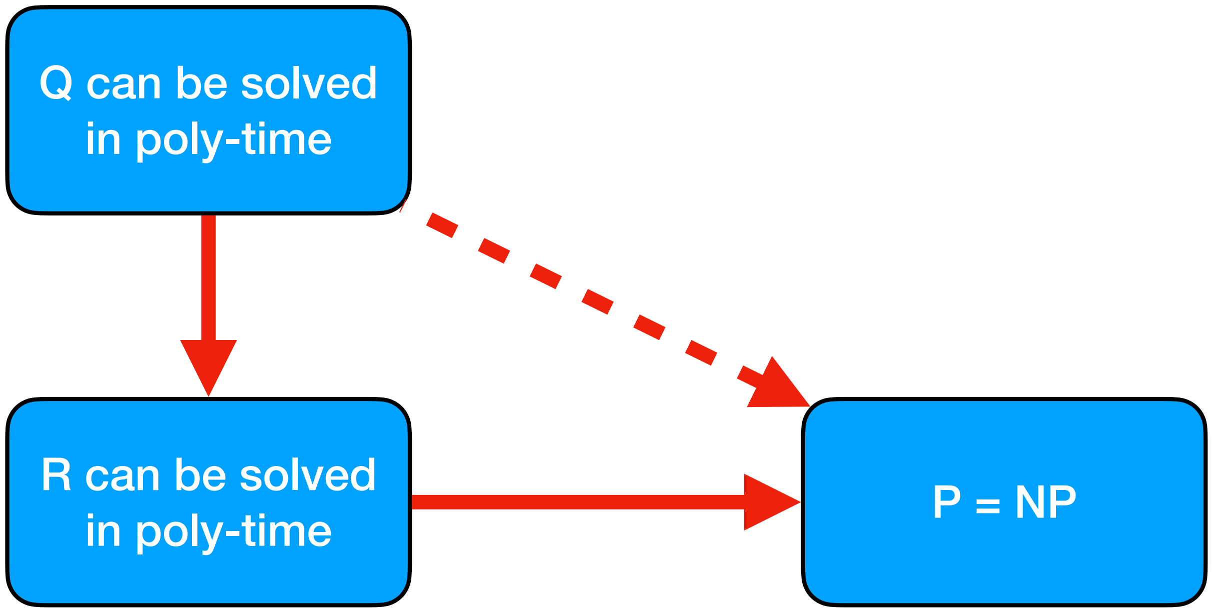
Plan



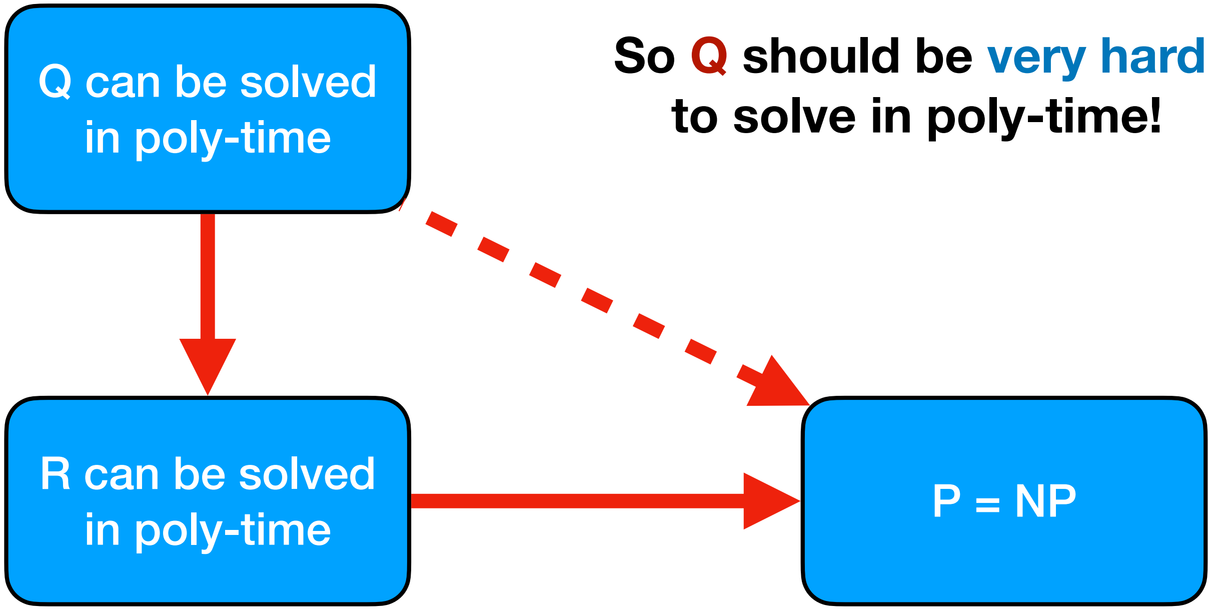
Plan



Plan



Plan



So Q should be very hard to solve in poly-time!

Plan

- **Goal**: Show that Q is very hard to solve in poly-time
- **Approach:**
  - Find any problem R such that if R can be solved in poly-time then P=NP
  - Show that R can be reduced to Q:
    - if Q can be solved in poly-time then R can also be solved in poly-time

NP-Hard Problems

- **NP-hard problems:**
  - We say a problem R is NP-hard if designing a poly-time algorithm for R implies P=NP

NP-Hard Problems

- **NP-hard problems:**
  - We say a problem R is NP-hard if designing a poly-time algorithm for R implies P=NP
- Note that the problem R itself may not be even in NP…
- This is not good since it means R can be too hard to begin with
- If R is too hard, then maybe even if we have a poly-time algorithm A for problem Q, we still cannot solve R with it in poly-time
- In other words, even if P=NP, there is no reason for R to have a poly-time algorithm

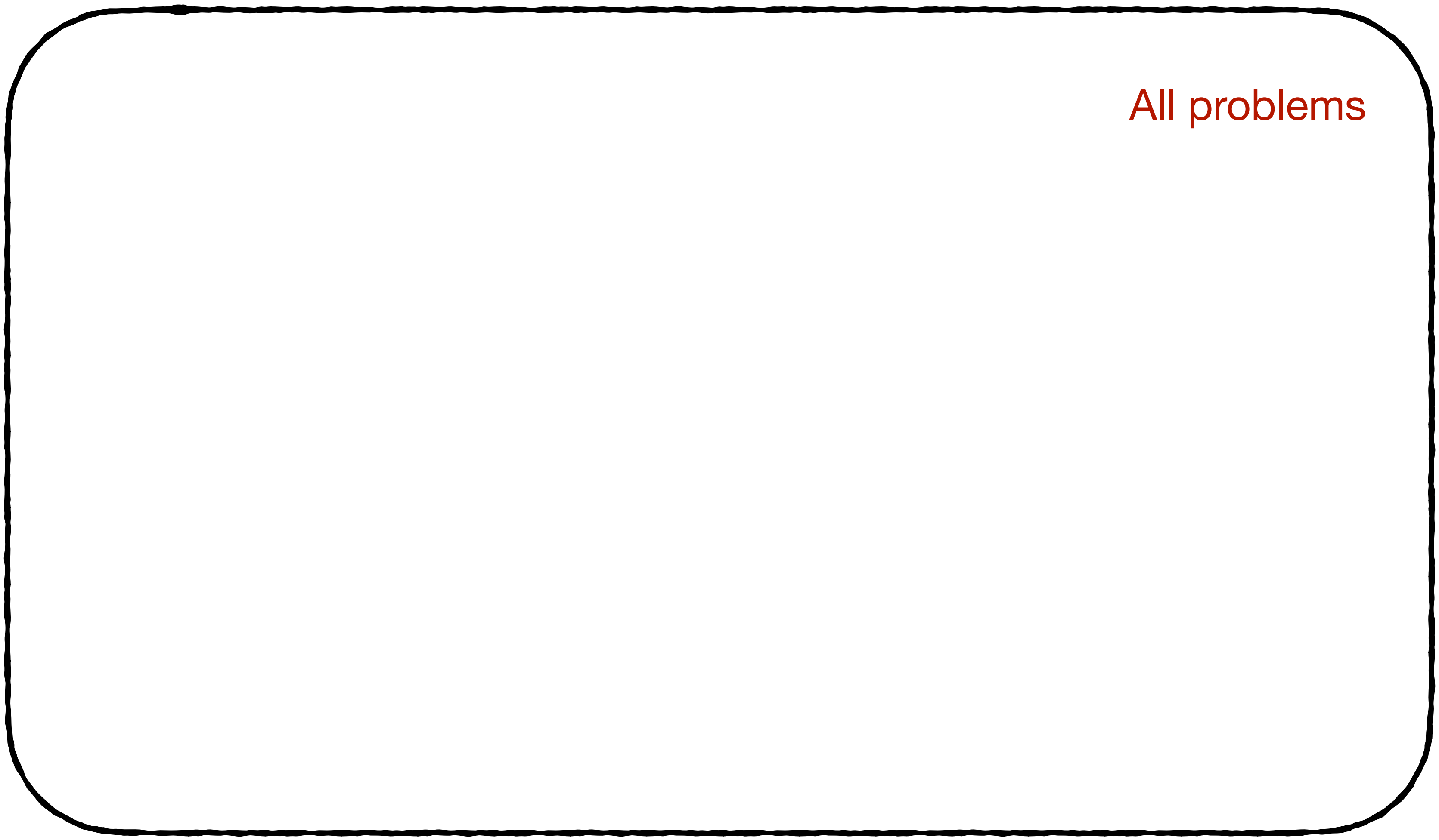
NP-Complete Problems

- **NP-complete problems:**
  - We say a problem R is NP-complete if (1) R is in NP itself, and (2) R is NP-hard

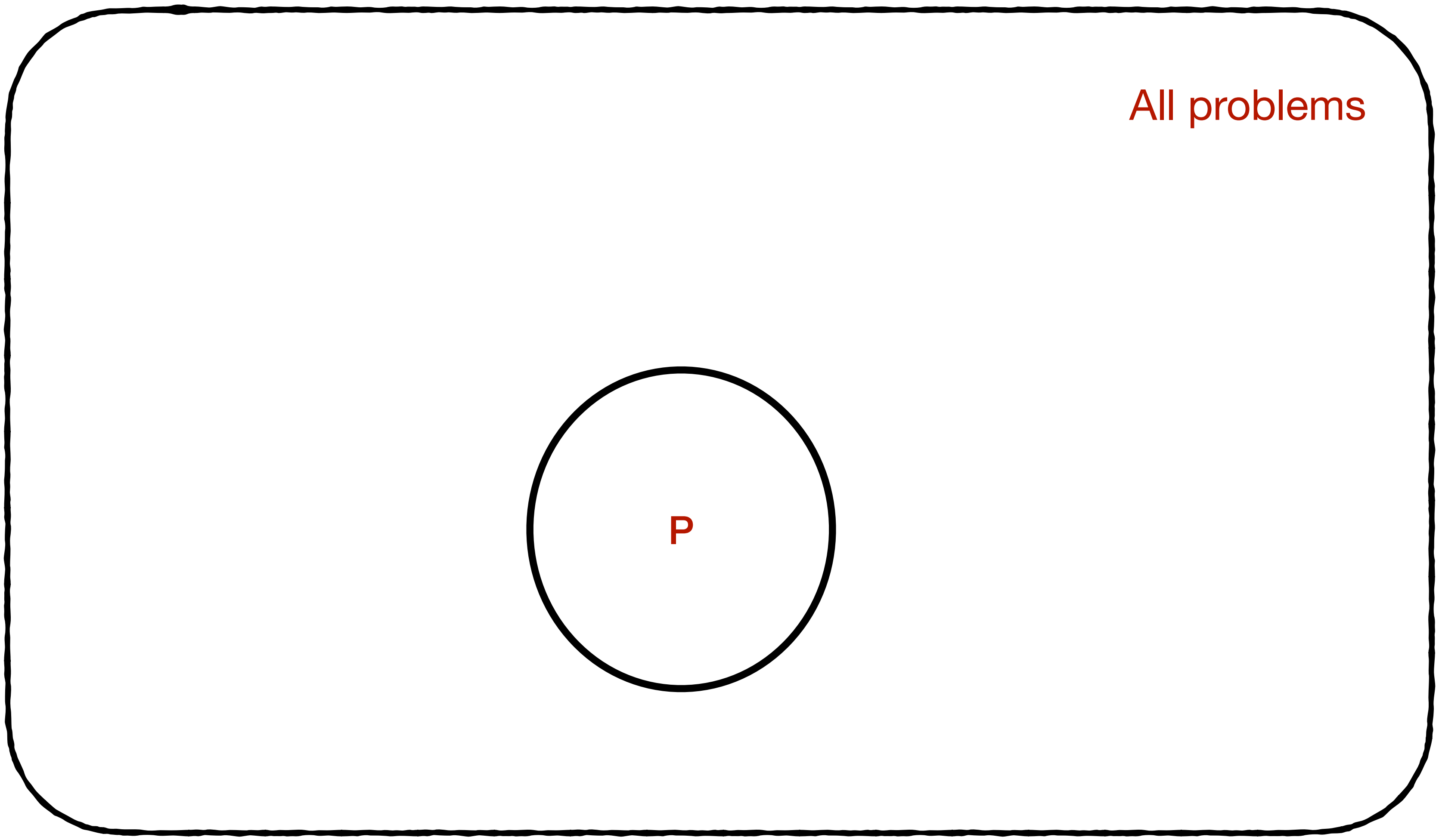
NP-Complete Problems

- **NP-complete problems:**
  - We say a problem R is NP-complete if (1) R is in NP itself, and (2) R is NP-hard
- Any NP-complete problem is in NP so we do not have the previous problem
  - If P=NP, then all NP-complete problems are solved in poly-time
- Any NP-complete problem is also in NP-hard (but the other direction is not necessarily true)

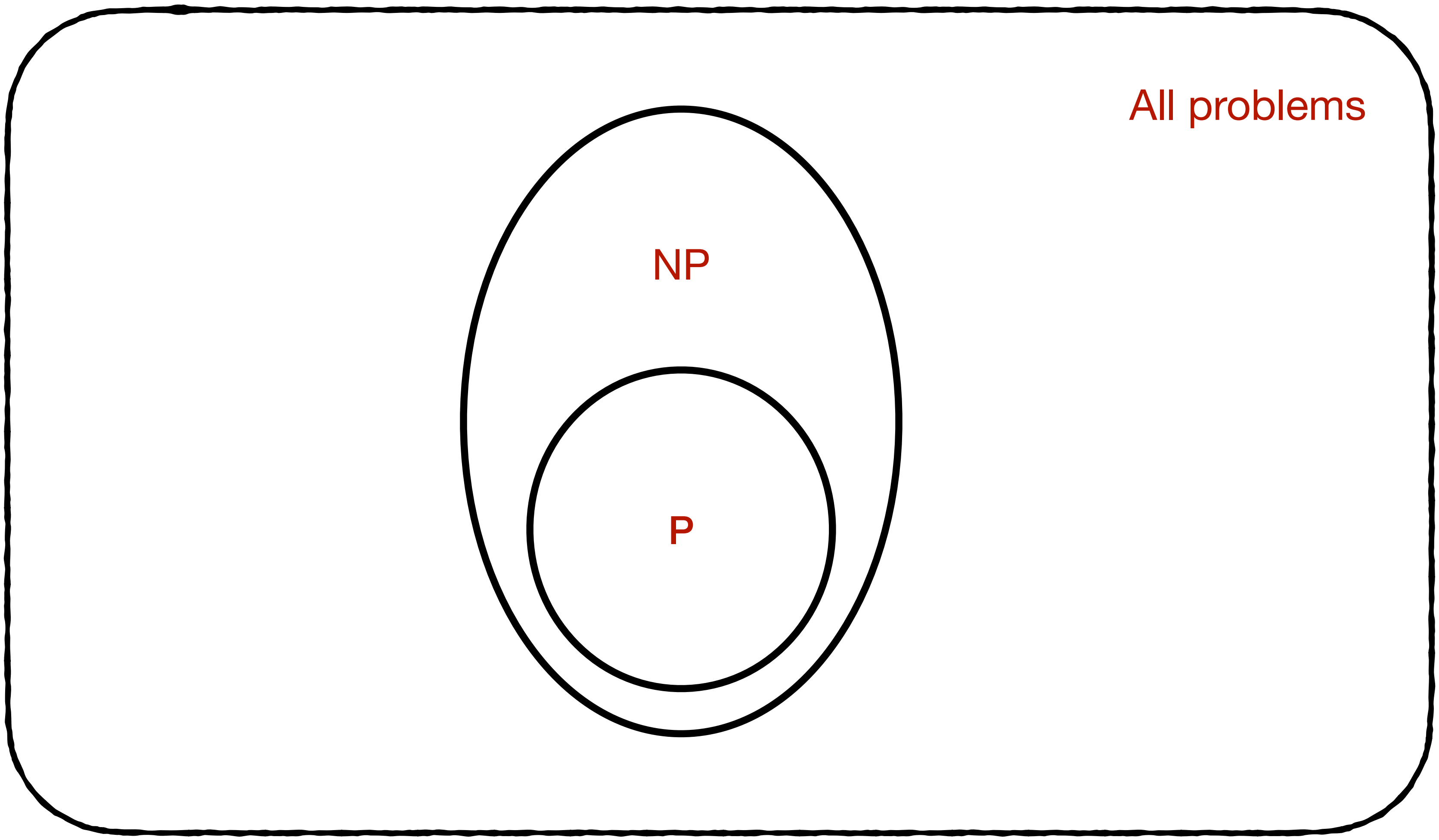
P, NP, NP-complete, NP-hard



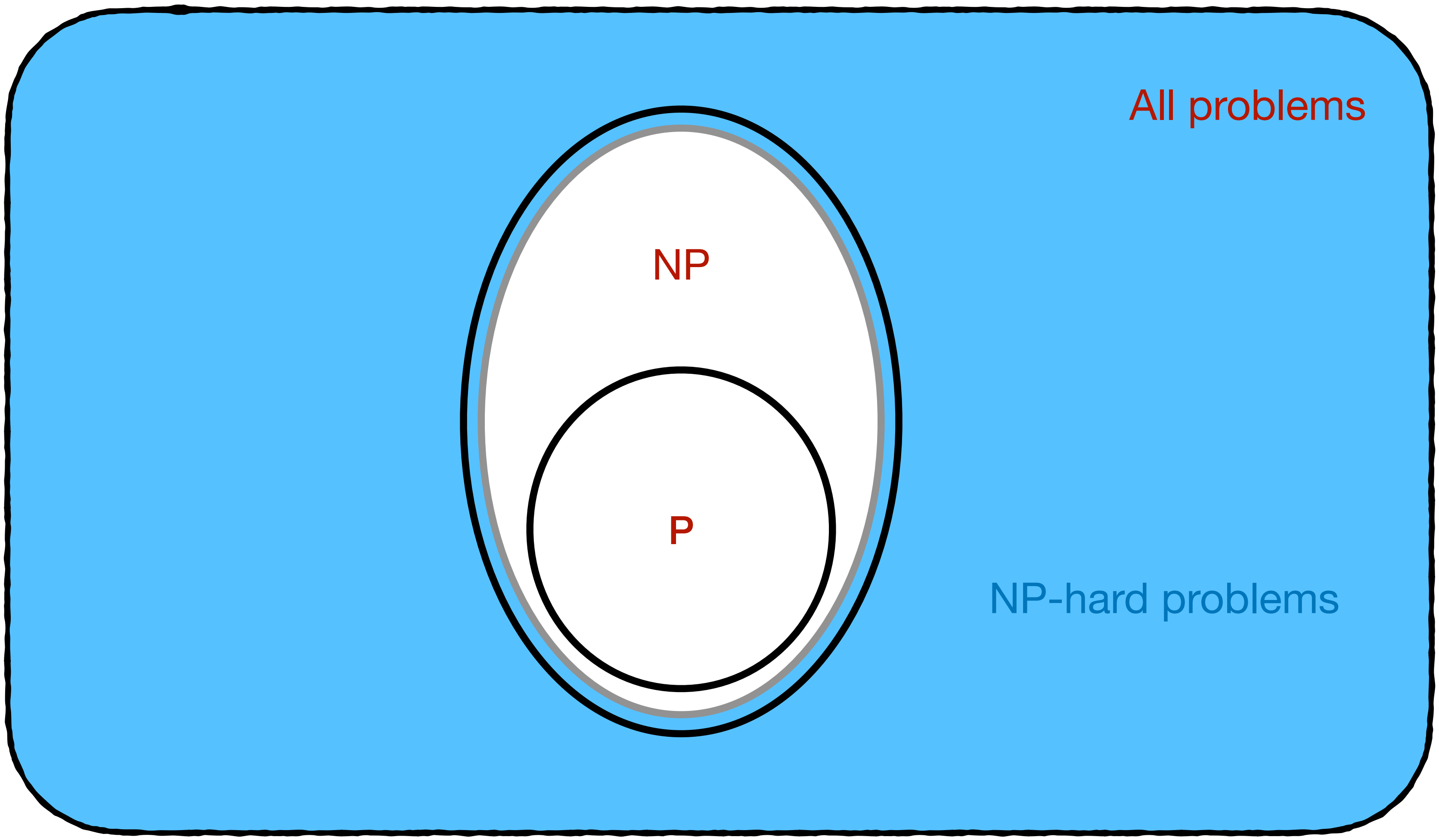
P, NP, NP-complete, NP-hard



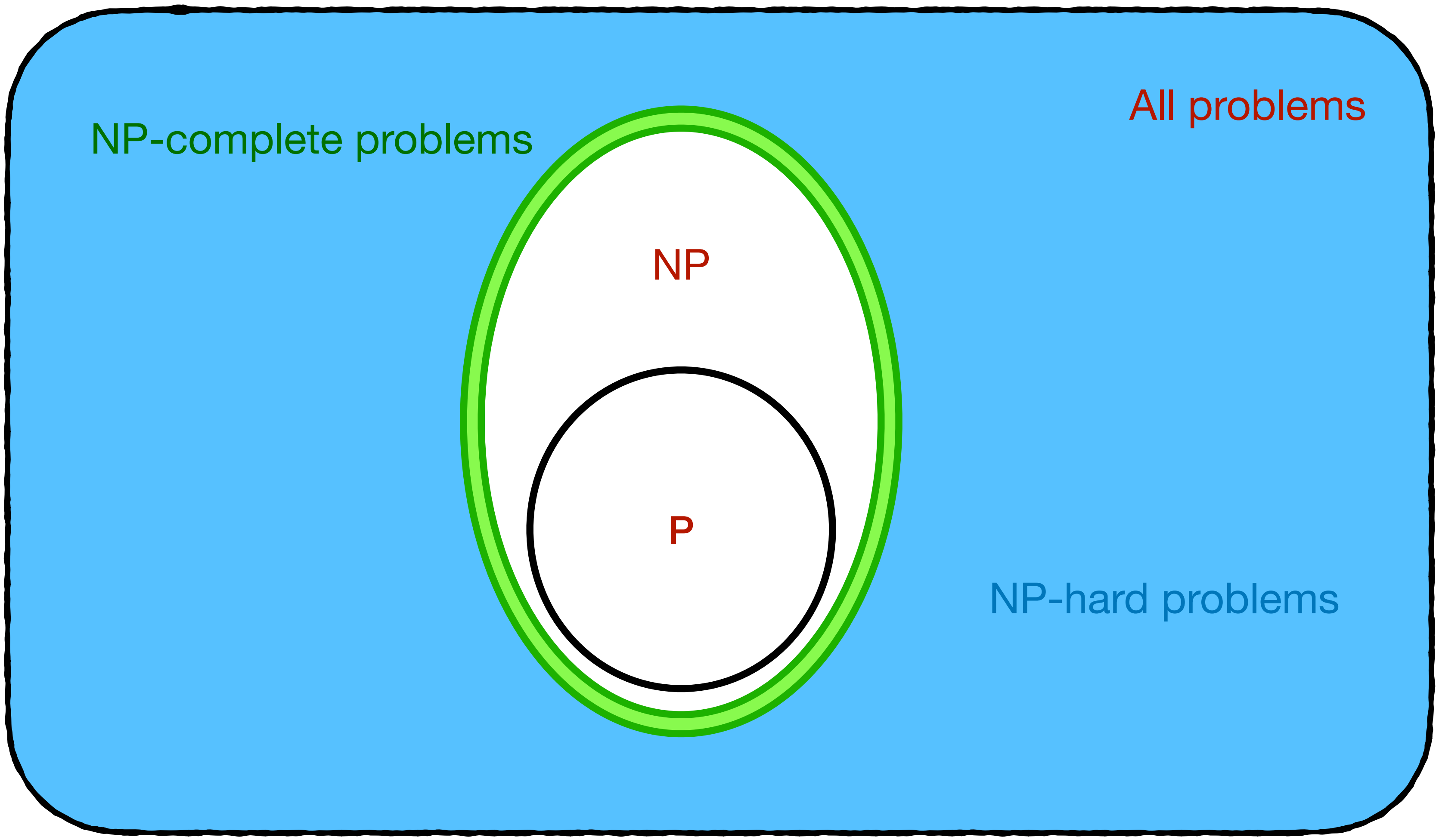
P, NP, NP-complete, NP-hard



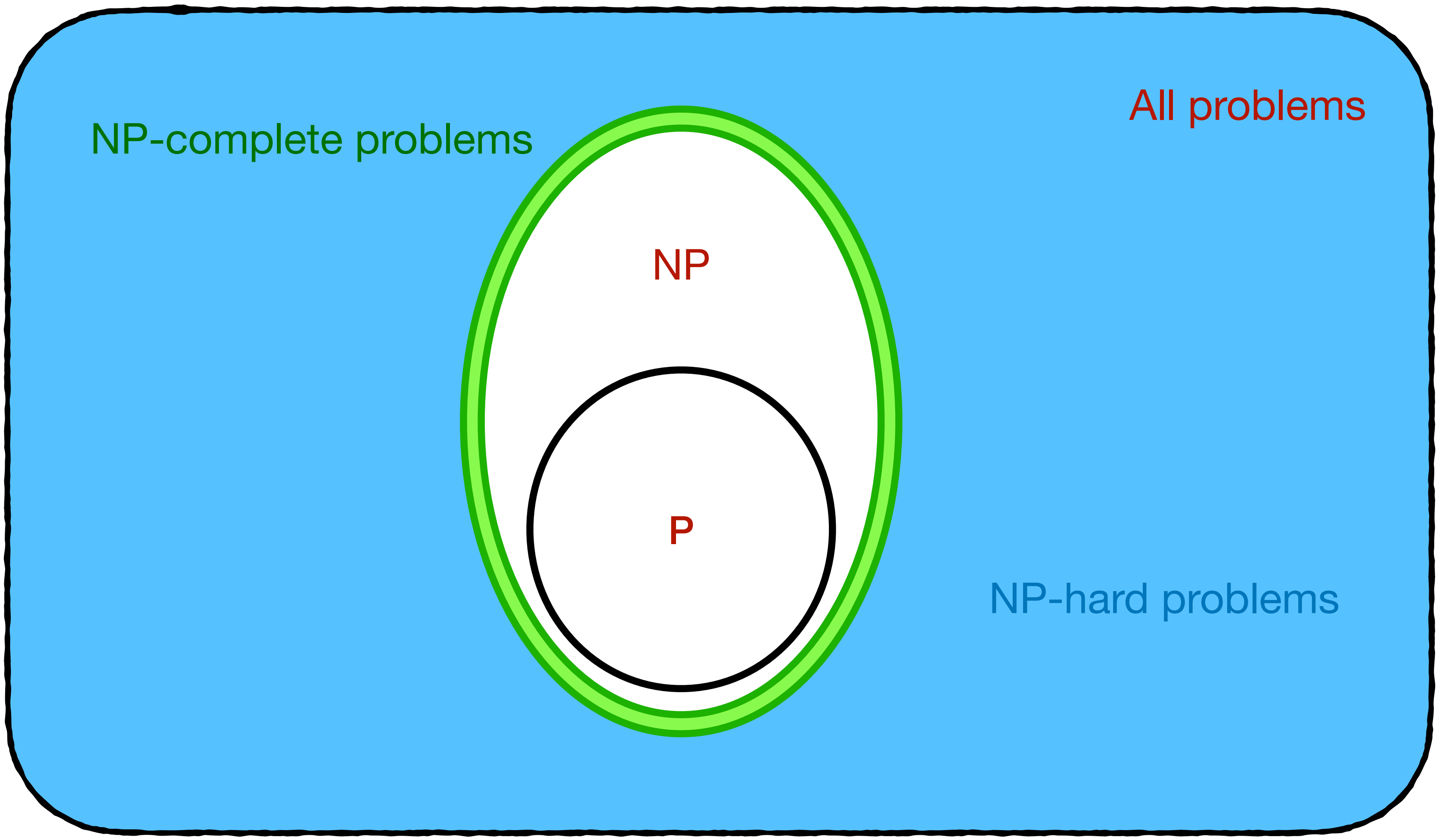
P, NP, NP-complete, NP-hard



P, NP, NP-complete, NP-hard

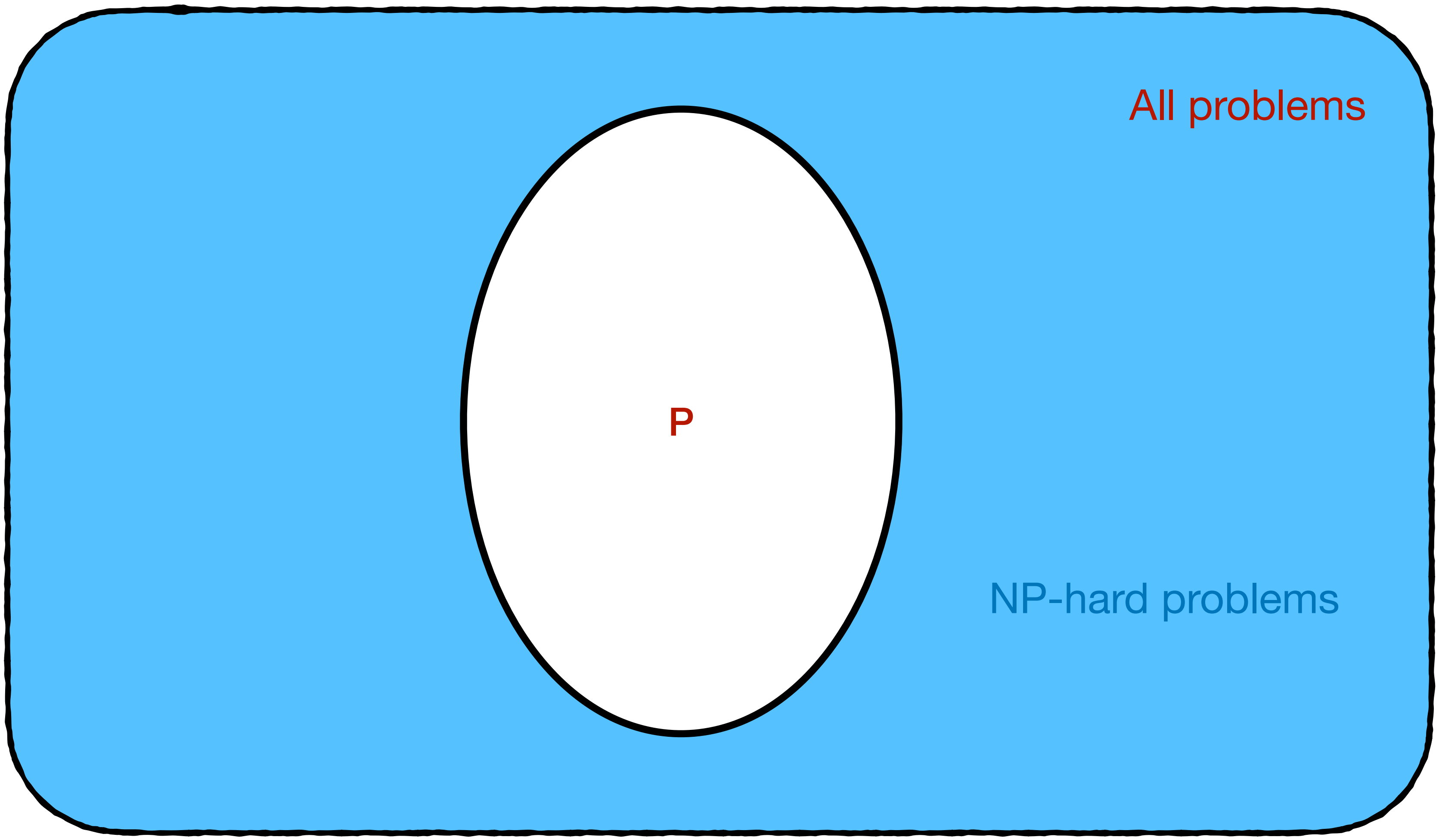


P, NP, NP-complete, NP-hard



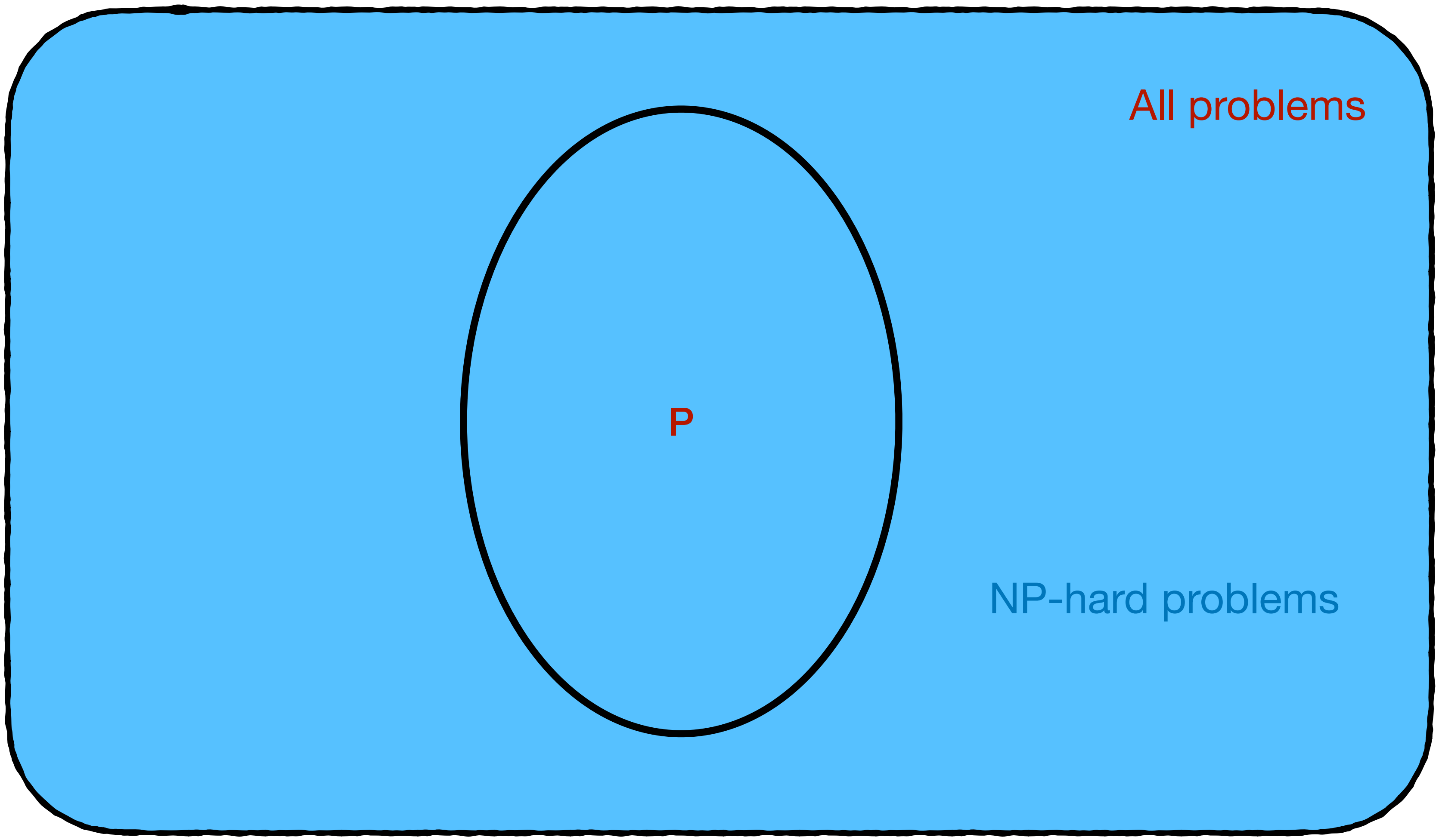
IF P = NP then this picture will become:

P, NP, NP-complete, NP-hard



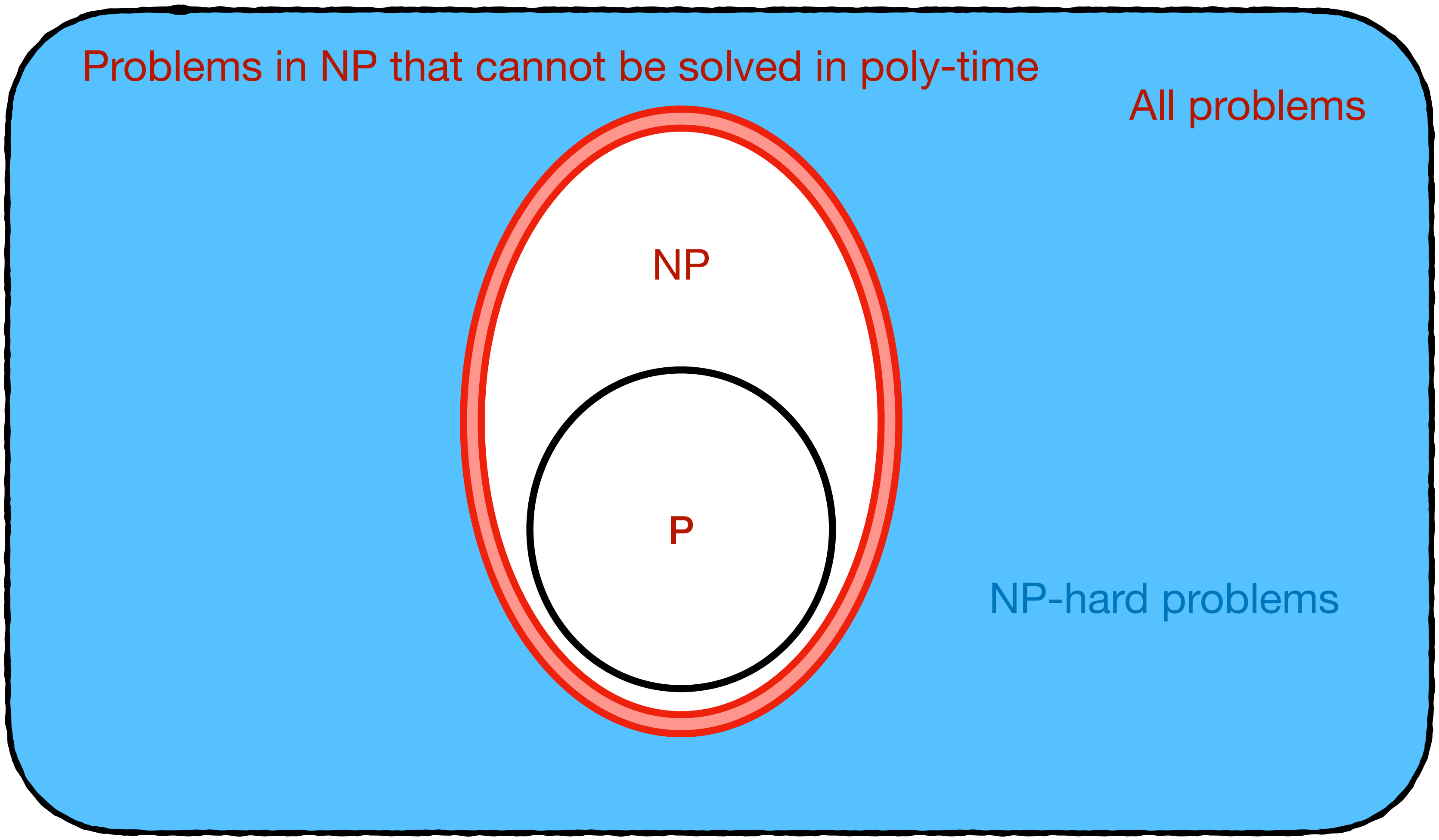
IF P = NP then this picture will become:

P, NP, NP-complete, NP-hard



IF P = NP then this picture will become: every problem will be NP-hard by definition

P, NP, NP-complete, NP-hard



IF P $\neq$ NP then this picture will become:

Plan

- **Goal**: Show that Q is very hard to solve in poly-time
- **Approach:**
  - Find any problem R such that if R can be solved in poly-time then P=NP
  - Show that R can be reduced to Q:
    - if Q can be solved in poly-time then R can also be solved in poly-time

Plan

- **Goal**: Show that Q is ~~very hard to solve in poly-time~~

  is NP-hard

- **Approach:**

  - Find any problem R ~~such that if R can be solved in poly-time then P=NP~~ **which is already known to be NP-hard**

  - Show that R can be reduced to Q:

    - if Q can be solved in poly-time then R can also be solved in poly-time

Plan

- **Goal**: Show that Q is ~~very hard to solve in poly-time~~ **is NP-hard**
- **Approach:**
  - Find any problem R ~~such that if R can be solved in poly-time then P=NP~~ **which is already known to be NP-hard**
  - Show that R can be reduced to Q:
    - if Q can be solved in poly-time then R can also be solved in poly-time
- If you want to prove R is NP-complete:
  - Prove that it is also in NP: give a poly-time verifier for it

Circuit-SAT problem & Cook-Levin Theorem

Plan

- **Goal**: Show that Q is NP-hard
- **Approach:**
  - Find any problem R which is already known to be NP-hard
  - Show that R can be reduced to Q:
    - if Q can be solved in poly-time then R can also be solved in poly-time

Plan

- **Goal**: Show that Q is NP-hard
- **Approach:**
  - Find any problem R which is already known to be NP-hard
  - Show that R can be reduced to Q:
    - if Q can be solved in poly-time then R can also be solved in poly-time
- For this plan to work we should have at least one NP-hard problem to begin with

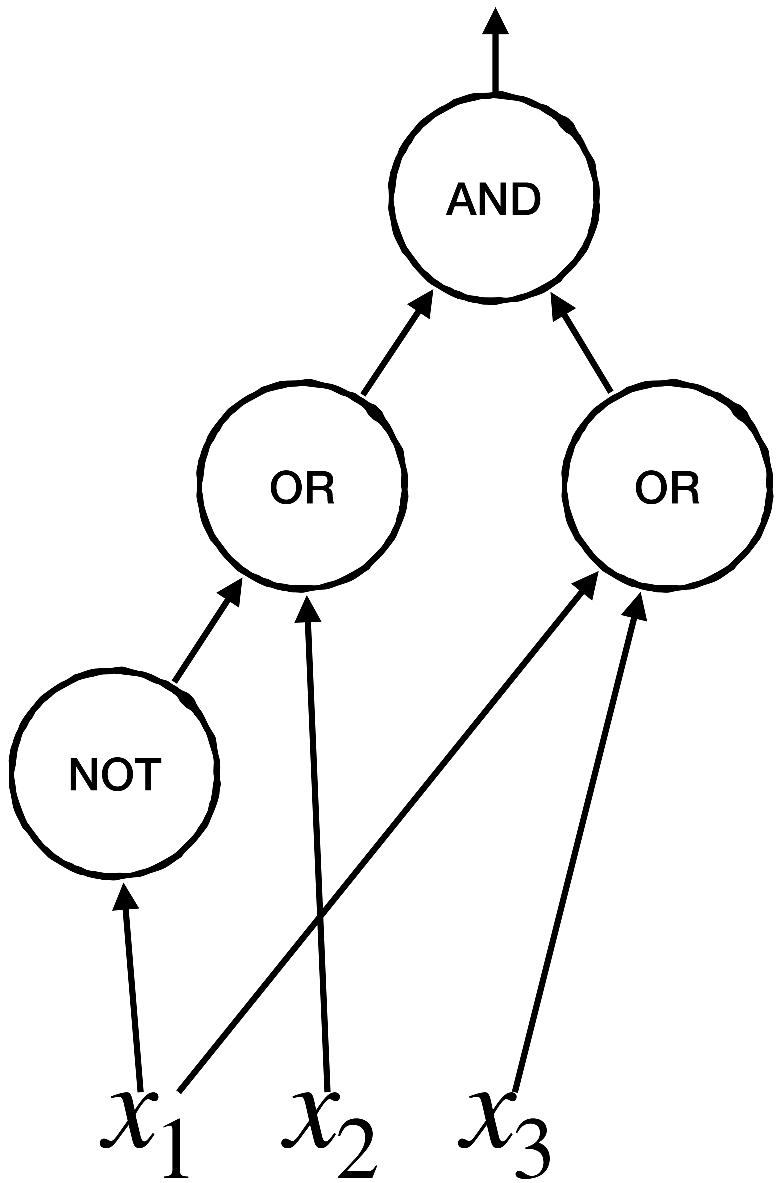
Circuit-SAT Problem

- The very first NP-hard problem is called the circuit-satisfiability problem or **circuit-SAT**

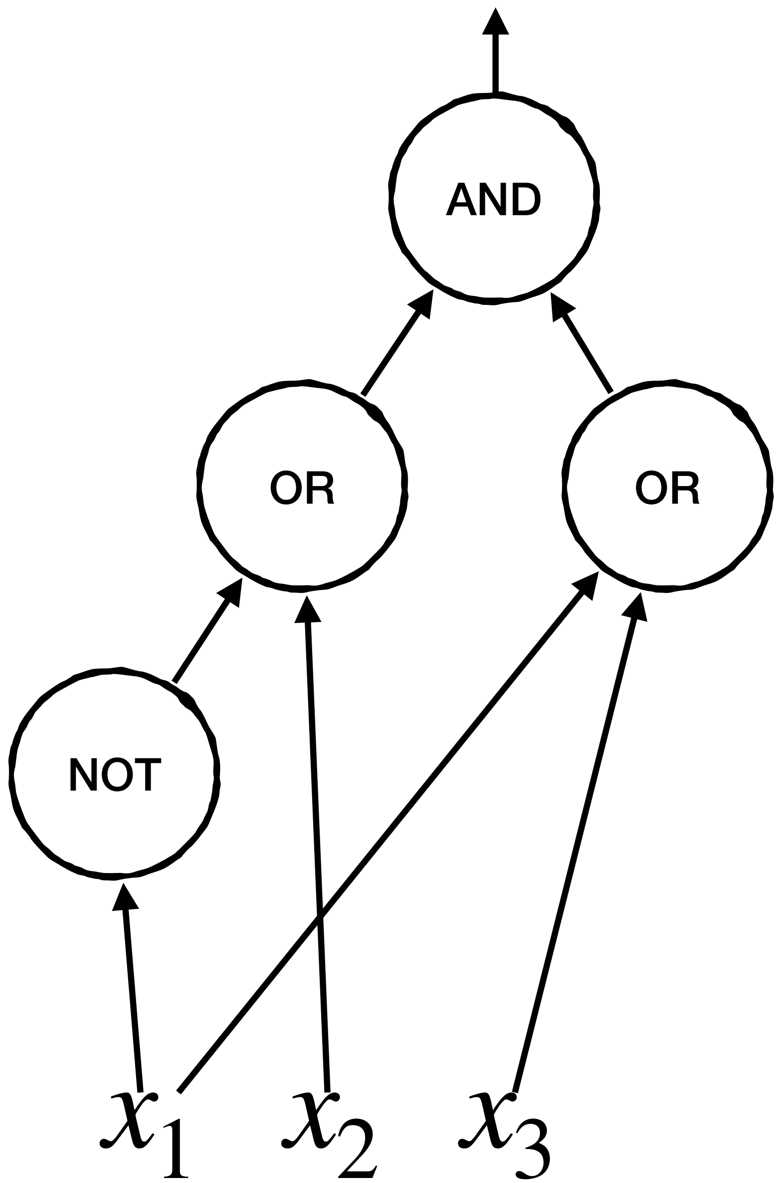
Circuit-SAT Problem

- The very first NP-hard problem is called the circuit-satisfiability problem or **circuit-SAT**
- **Input:**
  - A circuit C with binary AND and OR gates and unary NEGATE gates with n inputs in total
  - For any $x \in \{0,1\}^n$ we use C(x) to denote the value of circuit on the input x
- **Output:**
  - Is there any x such that C(x) = True?

Circuit-SAT Problem: Example

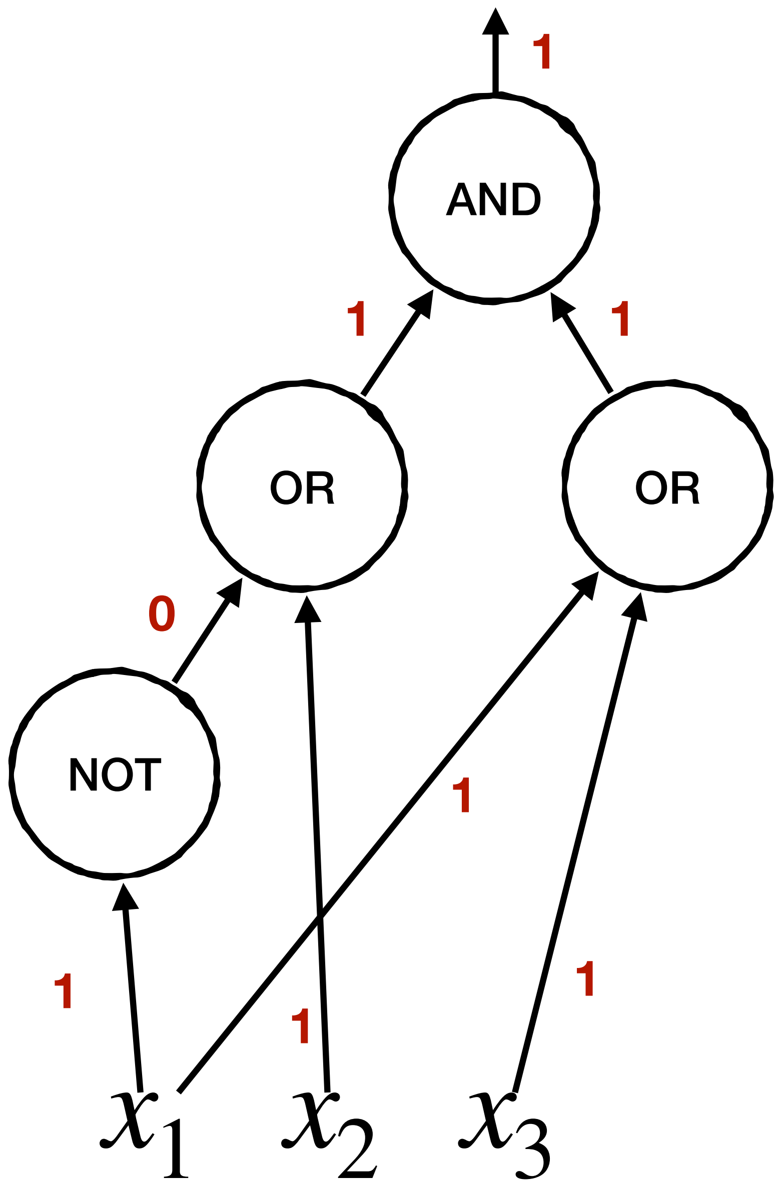


Circuit-SAT Problem: Example



Answer is YES

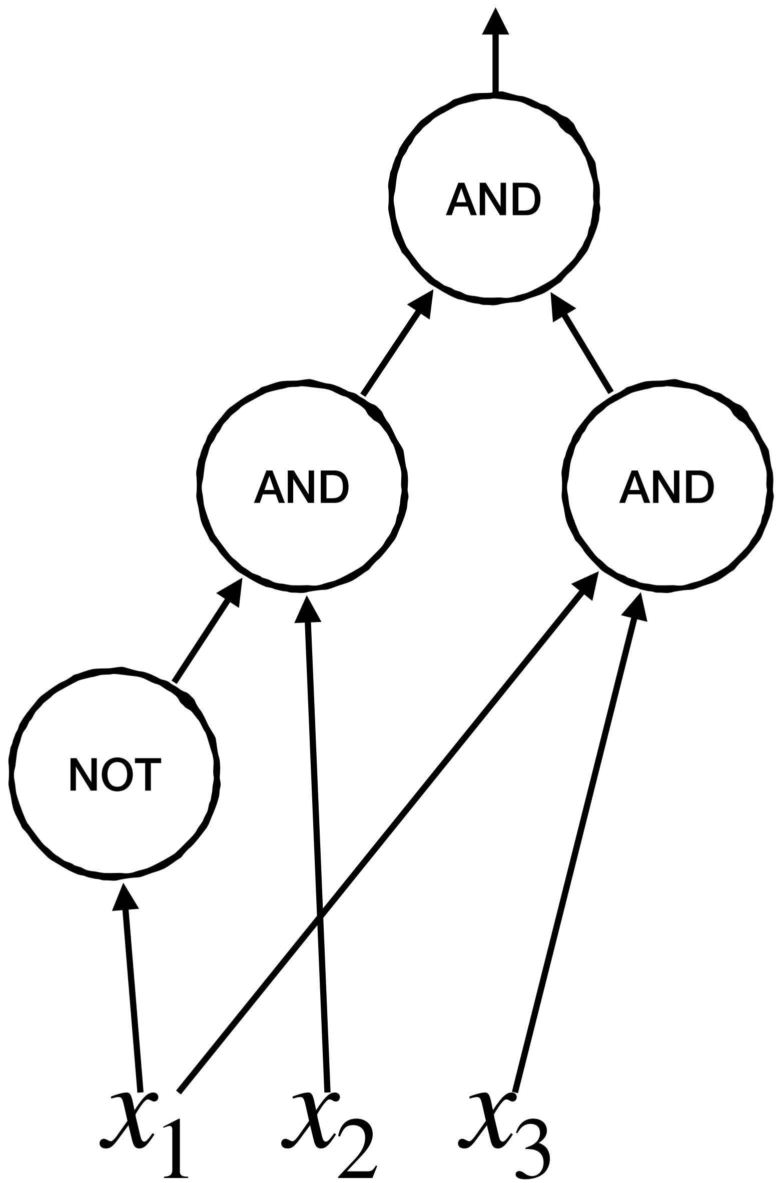
Circuit-SAT Problem: Example



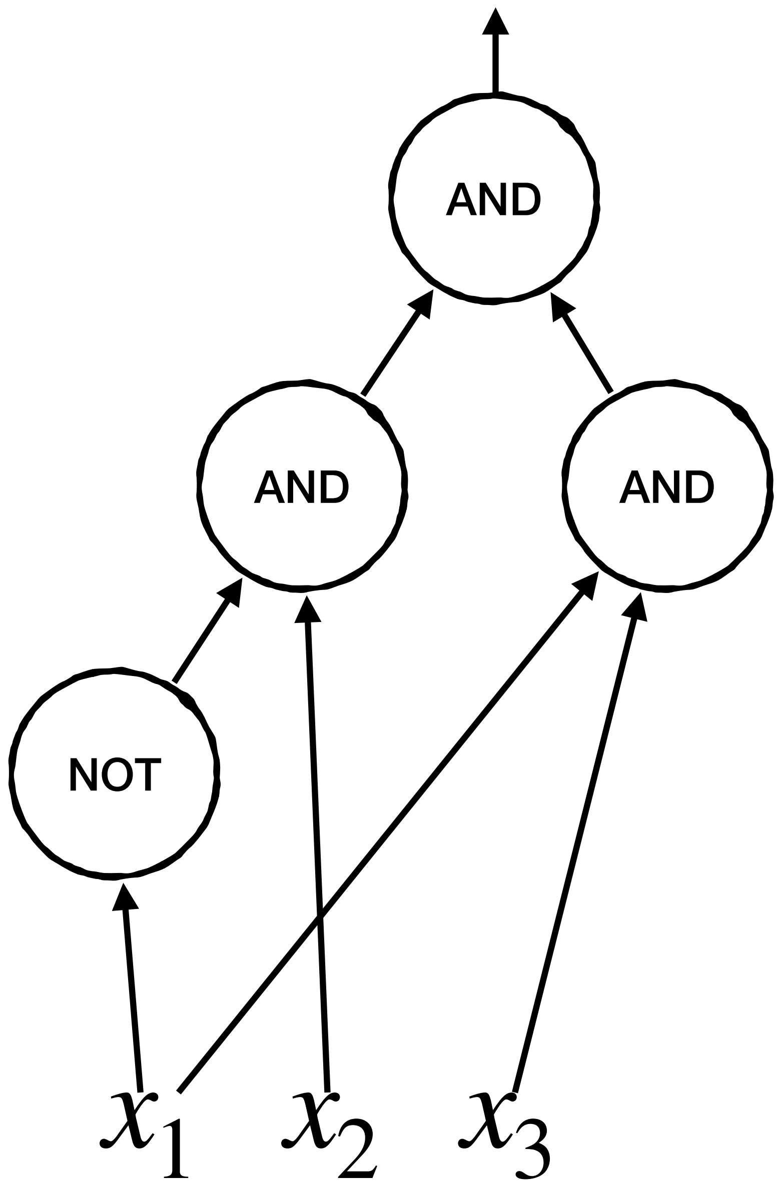
Answer is YES

For instance (1,1,1)

Circuit-SAT Problem: Example

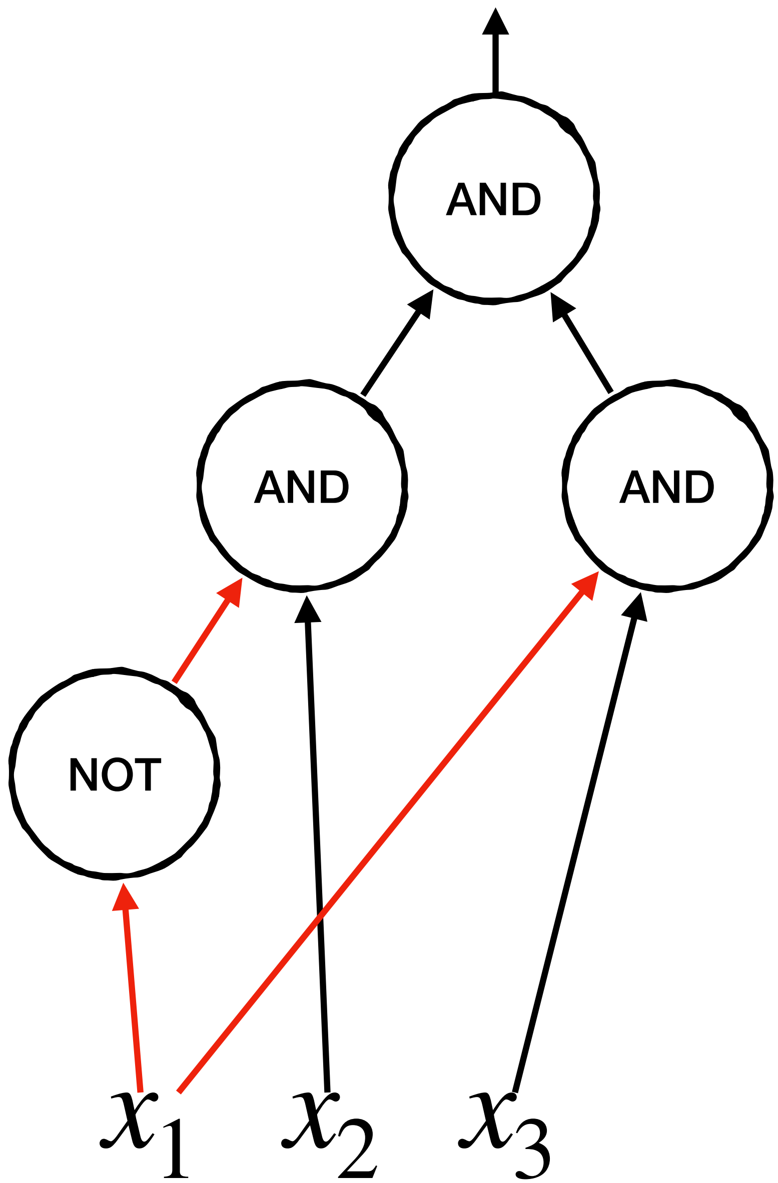


Circuit-SAT Problem: Example



Answer is NO

Circuit-SAT Problem: Example



Answer is NO

Circuit-SAT

- Circuit-SAT is in NP

Circuit-SAT

- Circuit-SAT is in NP
- A poly-time verifier:
  - The proof is an assignment x such that C(x) = 1
  - We can evaluate x in C to compute the answer — the evaluation is done bottom-up by computing value of each gate

Circuit-SAT

- Circuit-SAT is in NP
- A poly-time verifier:
  - The proof is an assignment x such that $C(x) = 1$
  - We can evaluate x in C to compute the answer — the evaluation is done bottom-up by computing value of each gate
- **Cook-Levin Theorem:** Circuit-SAT is NP-complete

Reductions

- We now know that circuit-SAT is NP-complete (and so NP-hard)
- We can use circuit-SAT in reductions to prove other problems are also NP-hard

Reductions

- We now know that circuit-SAT is NP-complete (and so NP-hard)
- We can use circuit-SAT in reductions to prove other problems are also NP-hard
- This is the topic of the next lecture