

# Basic Elements of Complexity Theory

Antonio Carzaniga

Faculty of Informatics  
Università della Svizzera italiana

May 23, 2024

- Basic complexity classes
- Polynomial reductions
- NP-completeness



- A ***polynomial-time algorithm*** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$

- A ***polynomial-time algorithm*** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$  polynomial-time?

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$  polynomial-time?

---

$$T(n) = n^2$$

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
--------	------------------

$T(n) = n^2$	Yes
--------------	-----

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
--------	------------------

$T(n) = n^2$	Yes
--------------	-----

$T(n) = n^3 - 2n^2 - 5$	
-------------------------	--

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No
$T(n) = n^7 + 7^{-n}$	

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No
$T(n) = n^7 + 7^{-n}$	Yes

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No
$T(n) = n^7 + 7^{-n}$	Yes
$T(n) = 5$	

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No
$T(n) = n^7 + 7^{-n}$	Yes
$T(n) = 5$	Yes

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No
$T(n) = n^7 + 7^{-n}$	Yes
$T(n) = 5$	Yes
$T(n) = n^{-7} \cdot 2^{n/7}$	

- A **polynomial-time algorithm** is one whose worst-case running time  $T(n)$ , on an input of size  $n$  bits, is  $O(n^k)$  for some *constant*  $k$
- **Examples:** algorithm  $A$  has a running time  $T(n)$ ; is  $A$  a *polynomial-time* algorithm?

$T(n)$	polynomial-time?
$T(n) = n^2$	Yes
$T(n) = n^3 - 2n^2 - 5$	Yes
$T(n) = \sqrt{n!}$	No
$T(n) = n^7 + 7^n$	No
$T(n) = n^7 + 7^{-n}$	Yes
$T(n) = 5$	Yes
$T(n) = n^{-7} \cdot 2^{n/7}$	No

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
------------------	--------------------------------

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
------------------	--------------------------------

<b>FIND</b> (sequential)	
--------------------------	--

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	$O(n^2)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	$O(n^2)$
<b>HEAPSORT</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	$O(n^2)$
<b>HEAPSORT</b>	$O(n \log n)$

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	$O(n^2)$
<b>HEAPSORT</b>	$O(n \log n)$
<b>EDIT-DISTANCE</b>	

# Examples of Polynomial-Time Algorithms

<u>Algorithm</u>	<u>worst-case running time</u>
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	$O(n^2)$
<b>HEAPSORT</b>	$O(n \log n)$
<b>EDIT-DISTANCE</b>	$O(n^2)$

# Examples of Polynomial-Time Algorithms

Algorithm	worst-case running time
<b>FIND</b> (sequential)	$O(n)$
<b>BINARY-SEARCH</b>	$O(\log n)$
<b>TREE-MINIMUM</b>	$O(n)$
<b>RB-INSERT</b>	$O(\log n)$
<b>INORDER-TREE-WALK</b>	$O(n)$
<b>INSERTION-SORT</b>	$O(n^2)$
<b>HEAPSORT</b>	$O(n \log n)$
<b>EDIT-DISTANCE</b>	$O(n^2)$
...	

# Polynomial vs. Super-Polynomial: Examples

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects  
*all pairs*

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

*all pairs polynomial:*  $\Theta(n^2)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

*all pairs* **polynomial:**  $\Theta(n^2)$

*all triples*

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

*all pairs* polynomial:  $\Theta(n^2)$

*all triples* polynomial:  $\Theta(n^3)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

***all permutations***

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

***all permutations***

***super-polynomial:***  $\Theta(n!)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

***all permutations***

***super-polynomial:***  $\Theta(n!)$

- You have a graph over  $n$  vertexes

***all edges***

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

***all permutations***

***super-polynomial:***  $\Theta(n!)$

- You have a graph over  $n$  vertexes

***all edges*** ***polynomial:***  $\Theta(n^2)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

**all pairs polynomial:**  $\Theta(n^2)$

**all triples polynomial:**  $\Theta(n^3)$

**all  $k$ -tuples for a fixed  $k$  polynomial:**  $\Theta(n^k)$

**all subsets**

**super-polynomial:**  $\Theta(2^n)$

**all permutations**

**super-polynomial:**  $\Theta(n!)$

- You have a graph over  $n$  vertexes

**all edges polynomial:**  $\Theta(n^2)$

**all trees**

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

***all permutations***

***super-polynomial:***  $\Theta(n!)$

- You have a graph over  $n$  vertexes

***all edges*** ***polynomial:***  $\Theta(n^2)$

***all trees***

***super-polynomial:***  $\Theta(n^{n-2})$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

***all pairs*** ***polynomial:***  $\Theta(n^2)$

***all triples*** ***polynomial:***  $\Theta(n^3)$

***all  $k$ -tuples*** for a fixed  $k$  ***polynomial:***  $\Theta(n^k)$

***all subsets***

***super-polynomial:***  $\Theta(2^n)$

***all permutations***

***super-polynomial:***  $\Theta(n!)$

- You have a graph over  $n$  vertexes

***all edges*** ***polynomial:***  $\Theta(n^2)$

***all trees***

***super-polynomial:***  $\Theta(n^{n-2})$

***all complete tours***

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

**all pairs polynomial:**  $\Theta(n^2)$

**all triples polynomial:**  $\Theta(n^3)$

**all  $k$ -tuples for a fixed  $k$  polynomial:**  $\Theta(n^k)$

**all subsets**

**super-polynomial:**  $\Theta(2^n)$

**all permutations**

**super-polynomial:**  $\Theta(n!)$

- You have a graph over  $n$  vertexes

**all edges polynomial:**  $\Theta(n^2)$

**all trees**

**super-polynomial:**  $\Theta(n^{n-2})$

**all complete tours**

**super-polynomial:**  $\Theta(n!)$

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

**all pairs polynomial:**  $\Theta(n^2)$

**all triples polynomial:**  $\Theta(n^3)$

**all  $k$ -tuples for a fixed  $k$  polynomial:**  $\Theta(n^k)$

**all subsets**

**super-polynomial:**  $\Theta(2^n)$

**all permutations**

**super-polynomial:**  $\Theta(n!)$

- You have a graph over  $n$  vertexes

**all edges polynomial:**  $\Theta(n^2)$

**all trees**

**super-polynomial:**  $\Theta(n^{n-2})$

**all complete tours**

**super-polynomial:**  $\Theta(n!)$

**all cuts**

# Polynomial vs. Super-Polynomial: Examples

- You have  $n$  objects

**all pairs polynomial:**  $\Theta(n^2)$

**all triples polynomial:**  $\Theta(n^3)$

**all  $k$ -tuples for a fixed  $k$  polynomial:**  $\Theta(n^k)$

**all subsets**

**super-polynomial:**  $\Theta(2^n)$

**all permutations**

**super-polynomial:**  $\Theta(n!)$

- You have a graph over  $n$  vertexes

**all edges polynomial:**  $\Theta(n^2)$

**all trees**

**super-polynomial:**  $\Theta(n^{n-2})$

**all complete tours**

**super-polynomial:**  $\Theta(n!)$

**all cuts**

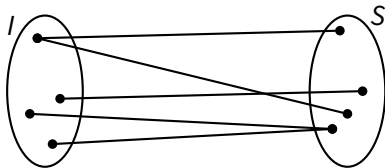
**super-polynomial:**  $\Theta(2^n)$

***polynomial*  $\equiv$  *good***

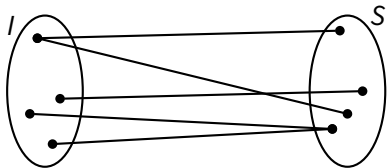
***super-polynomial*  $\equiv$  *bad***



- A **problem**  $Q$  is a binary relation between a set  $I$  of **instances** and a set  $S$  of **solutions**



- A **problem**  $Q$  is a binary relation between a set  $I$  of **instances** and a set  $S$  of **solutions**



- A **concrete problem**  $Q$  is one where  $I$  and  $S$  are the set of binary strings  $\{0, 1\}^*$ 
  - ▶ for all practical purposes, instances and solutions can be **encoded** as binary strings (i.e., mapped into  $\{0, 1\}^*$ )
  - ▶ we consider only sensible encodings...



- A ***decision problem***  $Q$  is one where the set of solutions is  $S = \{0, 1\}$

- A **decision problem**  $Q$  is one where the set of solutions is  $S = \{0, 1\}$

**Example:**

1	→	0
10	→	1
11	→	1
100	→	0
101	→	1
110	→	0
111	→	1
1000	→	0
1001	→	0
1010	→	0
1011	→	1
1100	→	0
1101	→	1
...		

- A **decision problem**  $Q$  is one where the set of solutions is  $S = \{0, 1\}$

**Example:**

1	→	0
10	→	1
11	→	1
100	→	0
101	→	1
110	→	0
111	→	1
1000	→	0
1001	→	0
1010	→	0
1011	→	1
1100	→	0
1101	→	1
...		

*Primality Testing*

# Decision vs. Optimization: Example

## Decision vs. Optimization: Example

- Shortest path in a graph

$$G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z \longrightarrow a, c, \dots, z$$

# Decision vs. Optimization: Example

- Shortest path in a graph

$G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z \longrightarrow a, c, \dots, z$

instance



# Decision vs. Optimization: Example

## ■ Shortest path in a graph

$$\underbrace{G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z}_{\text{instance}} \longrightarrow \underbrace{a, c, \dots, z}_{\text{solution}}$$

- ▶ *input*: a graph  $G$ , a source vertex ( $a$ ), and a destination vertex ( $z$ )
- ▶ *output*: a sequence of vertexes  $a, c, \dots, z$

# Decision vs. Optimization: Example

## ■ Shortest path in a graph

$$\underbrace{G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z}_{\text{instance}} \longrightarrow \underbrace{a, c, \dots, z}_{\text{solution}}$$

- ▶ *input*: a graph  $G$ , a source vertex ( $a$ ), and a destination vertex ( $z$ )
- ▶ *output*: a sequence of vertexes  $a, c, \dots, z$

## ■ Shortest path as a **decision problem**

$$G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z, 10 \longrightarrow 1$$

# Decision vs. Optimization: Example

## ■ Shortest path in a graph

$$\underbrace{G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z}_{\text{instance}} \longrightarrow \underbrace{a, c, \dots, z}_{\text{solution}}$$

- ▶ *input*: a graph  $G$ , a source vertex ( $a$ ), and a destination vertex ( $z$ )
- ▶ *output*: a sequence of vertexes  $a, c, \dots, z$

## ■ Shortest path as a **decision problem**

$$\underbrace{G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z, 10}_{\text{instance}} \longrightarrow 1$$

# Decision vs. Optimization: Example

## ■ Shortest path in a graph

$$\underbrace{G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z}_{\text{instance}} \longrightarrow \underbrace{a, c, \dots, z}_{\text{solution}}$$

- ▶ *input*: a graph  $G$ , a source vertex ( $a$ ), and a destination vertex ( $z$ )
- ▶ *output*: a sequence of vertexes  $a, c, \dots, z$

## ■ Shortest path as a **decision problem**

$$\underbrace{G = (V = \{a, b, c, \dots\}, E = \{(a, c), \dots\}), a, z, 10}_{\text{instance}} \longrightarrow \underbrace{1}_{\text{solution}}$$

- ▶ *input*: a graph  $G$ , a start vertex ( $a$ ), an end vertex ( $z$ ), and a path length (10)
- ▶ *output*: 1 if there is a path of (at most) the given length

# Decision vs. Optimization

- *We focus on decision problems only*

# Decision vs. Optimization

- *We focus on decision problems only*
- An optimization problem is *at least as hard* as its corresponding decision problem
  - ▶ having a solution to the optimization gives an immediate solution to the decision problem

# Decision vs. Optimization

- *We focus on decision problems only*
- An optimization problem is ***at least as hard*** as its corresponding decision problem
  - ▶ having a solution to the optimization gives an immediate solution to the decision problem
- An optimization problem is ***not much harder*** than the corresponding decision problem

# Decision vs. Optimization

- *We focus on decision problems only*
- An optimization problem is ***at least as hard*** as its corresponding decision problem
  - ▶ having a solution to the optimization gives an immediate solution to the decision problem
- An optimization problem is ***not much harder*** than the corresponding decision problem
  - ▶ having a solution to the decision problem does not give an immediate solution to the optimization problem
  - ▶ but we can typically use the decision problem as a subroutine in some kind of (binary) search to solve the corresponding optimization problem

# The Complexity Class P

# The Complexity Class P

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

- Examples

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

- Examples

- ▶ shortest path (decision variant)

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

- Examples

- ▶ shortest path (decision variant)—Dijkstra's algorithm

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

- Examples

- ▶ shortest path (decision variant)—Dijkstra's algorithm
- ▶ primality

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

## ■ Examples

- ▶ shortest path (decision variant)—Dijkstra's algorithm
- ▶ primality—a relatively recent theoretical result...
  - ▶ in 2002: Agrawal, Kayal, and Saxena from IIT Kanpur
  - ▶ *Neeraj Kayal and Nitin Saxena were Bachelor students!*

- A concrete decision problem  $Q$  is **polynomial-time solvable** if there is a polynomial-time algorithm  $A$  that solves it

The **complexity class P** is the set of all concrete decision problems that are **polynomial-time solvable**

problem  $Q \in P \iff \exists$  algorithm  $A, A$  solves  $Q$  in polynomial time (worst case)

## ■ Examples

- ▶ shortest path (decision variant)—Dijkstra's algorithm
- ▶ primality—a relatively recent theoretical result...
  - ▶ in 2002: Agrawal, Kayal, and Saxena from IIT Kanpur
  - ▶ *Neeraj Kayal and Nitin Saxena were Bachelor students!*
- ▶ parsing a Java program
- ▶ ...

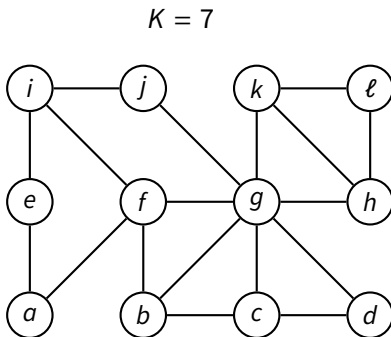
**Verifying is Easy**

■ **Example:** *Vertex cover* (decision variant)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise

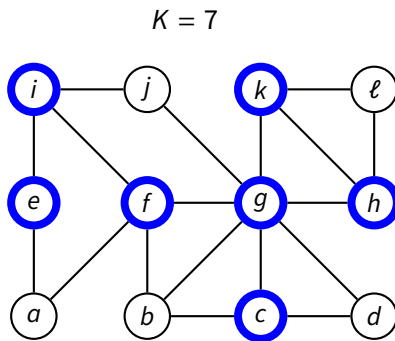
■ **Example:** *Vertex cover* (decision variant)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise



■ **Example: Vertex cover** (decision variant)

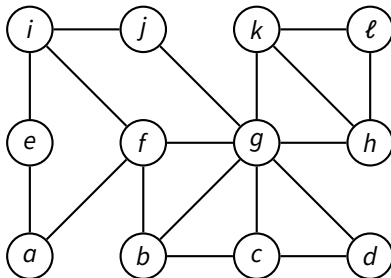
- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E, u \in S$  or  $v \in S$  (or both); 0 otherwise



■ **Example: Vertex cover** (decision variant)

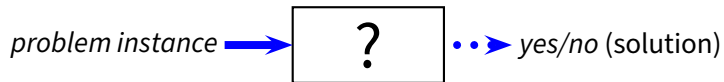
- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise

$K = 6?$



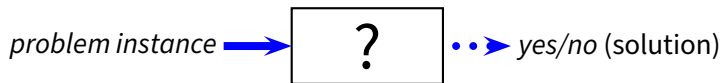
# Polynomial-Time Verification

- We might not know how to *solve* a problem in polynomial time

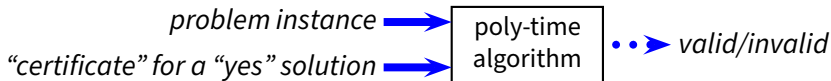


# Polynomial-Time Verification

- We might not know how to *solve* a problem in polynomial time

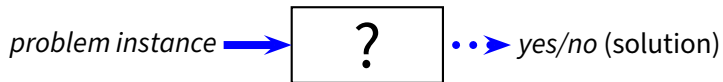


- But we might know how to **verify a given solution** in polynomial time

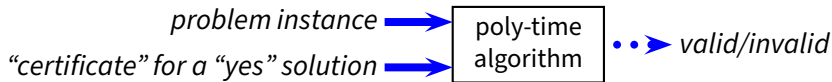


# Polynomial-Time Verification

- We might not know how to *solve* a problem in polynomial time



- But we might know how to ***verify a given solution*** in polynomial time



- Examples

- ▶ longest path (decision variant)
- ▶ knapsack (decision variant)

# The Complexity Class NP

- A concrete decision problem  $Q$  is **polynomial-time verifiable** if
  - ▶ there is a polynomial-time algorithm  $A$
  - ▶ for each instance  $x \in I$  that has a “yes” solution ( $Q(x) = 1$ )
  - ▶ there is a **certificate**  $y$  of polynomial size  $|y| = O(|x|^c)$ , for some constant  $c$
  - ▶ such that  $A(x, y) = 1$

■ A concrete decision problem  $Q$  is **polynomial-time verifiable** if

- ▶ there is a polynomial-time algorithm  $A$
- ▶ for each instance  $x \in I$  that has a “yes” solution ( $Q(x) = 1$ )
- ▶ there is a **certificate**  $y$  of polynomial size  $|y| = O(|x|^c)$ , for some constant  $c$
- ▶ such that  $A(x, y) = 1$

$A(x, y)$  **verifies in polynomial time that  $y$  proves that  $Q(x) = 1$**

- A concrete decision problem  $Q$  is **polynomial-time verifiable** if
  - ▶ there is a polynomial-time algorithm  $A$
  - ▶ for each instance  $x \in I$  that has a “yes” solution ( $Q(x) = 1$ )
  - ▶ there is a **certificate**  $y$  of polynomial size  $|y| = O(|x|^c)$ , for some constant  $c$
  - ▶ such that  $A(x, y) = 1$

$A(x, y)$  **verifies in polynomial time that  $y$  proves that  $Q(x) = 1$**

The **complexity class NP** is the set of all concrete decision problems that are **polynomial-time verifiable**

# The Complexity Class NP

- A concrete decision problem  $Q$  is **polynomial-time verifiable** if
  - ▶ there is a polynomial-time algorithm  $A$
  - ▶ for each instance  $x \in I$  that has a “yes” solution ( $Q(x) = 1$ )
  - ▶ there is a **certificate**  $y$  of polynomial size  $|y| = O(|x|^c)$ , for some constant  $c$
  - ▶ such that  $A(x, y) = 1$

$A(x, y)$  **verifies in polynomial time that  $y$  proves that  $Q(x) = 1$**

The **complexity class NP** is the set of all concrete decision problems that are **polynomial-time verifiable**

- **NP does not mean non-polynomial!**

- A concrete decision problem  $Q$  is **polynomial-time verifiable** if
  - ▶ there is a polynomial-time algorithm  $A$
  - ▶ for each instance  $x \in I$  that has a “yes” solution ( $Q(x) = 1$ )
  - ▶ there is a **certificate**  $y$  of polynomial size  $|y| = O(|x|^c)$ , for some constant  $c$
  - ▶ such that  $A(x, y) = 1$

$A(x, y)$  **verifies in polynomial time that  $y$  proves that  $Q(x) = 1$**

The **complexity class NP** is the set of all concrete decision problems that are **polynomial-time verifiable**

- **NP does not mean non-polynomial!**
  - ▶ it means “non-deterministic polynomial”

- A concrete decision problem  $Q$  is **polynomial-time verifiable** if
  - ▶ there is a polynomial-time algorithm  $A$
  - ▶ for each instance  $x \in I$  that has a “yes” solution ( $Q(x) = 1$ )
  - ▶ there is a **certificate**  $y$  of polynomial size  $|y| = O(|x|^c)$ , for some constant  $c$
  - ▶ such that  $A(x, y) = 1$

$A(x, y)$  **verifies in polynomial time that  $y$  proves that  $Q(x) = 1$**

The **complexity class NP** is the set of all concrete decision problems that are **polynomial-time verifiable**

- **NP does not mean non-polynomial!**
  - ▶ it means “non-deterministic polynomial”
- *polynomial-time solvable*  $\implies$  *polynomial-time verifiable*

$$P \subseteq NP$$

# The Big Open Question

- *polynomial-time verifiable*  $\stackrel{?}{\implies}$  *polynomial-time solvable*

- *polynomial-time verifiable*  $\stackrel{?}{\implies}$  *polynomial-time solvable*
- Or are there problems for which there is a polynomial-time verification algorithm but there are no polynomial-time algorithms to find solutions?

- *polynomial-time verifiable*  $\stackrel{?}{\implies}$  *polynomial-time solvable*
- Or are there problems for which there is a polynomial-time verification algorithm but there are no polynomial-time algorithms to find solutions?

P = NP?

- *polynomial-time verifiable*  $\stackrel{?}{\implies}$  *polynomial-time solvable*
- Or are there problems for which there is a polynomial-time verification algorithm but there are no polynomial-time algorithms to find solutions?

P = NP?

- Most theoretical computing scientists *believe* that  $P \neq NP$

# The Big Open Question

- *polynomial-time verifiable*  $\stackrel{?}{\implies}$  *polynomial-time solvable*
- Or are there problems for which there is a polynomial-time verification algorithm but there are no polynomial-time algorithms to find solutions?

P = NP?

- Most theoretical computing scientists *believe* that  $P \neq NP$

***Finding a solution to a problem is believed to be inherently more difficult than verifying a given solution (or a proof of a solution)***

***...but nobody has been able to prove that this is the case!***



## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

- Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

- Examples

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y)$

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$
- ▶  $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z)$

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$
- ▶  $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z) \longrightarrow 0$

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$
- ▶  $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z) \longrightarrow 0$

## ■ SAT $\in$ NP?

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$
- ▶  $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z) \longrightarrow 0$

## ■ SAT $\in$ NP?

- ▶ yes: given an assignment that satisfies the formula, it is easy (poly-time) to verify that the formula is satisfiable

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$
- ▶  $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z) \longrightarrow 0$

## ■ SAT $\in$ NP?

- ▶ yes: given an assignment that satisfies the formula, it is easy (poly-time) to verify that the formula is satisfiable

## ■ SAT $\in$ P?

## ■ Satisfiability problem (SAT)

- ▶ *Input*: a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶ *Output*: 1 iff there is an assignment of variables that satisfies the formula

## ■ Examples

- ▶  $\neg x \wedge (\neg y \vee \neg z) \wedge \neg z \wedge (x \vee y) \longrightarrow 1 \quad (x = 0, y = 1, z = 0)$
- ▶  $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z) \longrightarrow 0$

## ■ SAT $\in$ NP?

- ▶ yes: given an assignment that satisfies the formula, it is easy (poly-time) to verify that the formula is satisfiable

## ■ SAT $\in$ P?

- ▶ we don't know

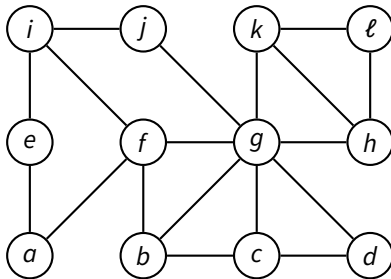
### ■ Vertex cover (VC)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise

## Example: Vertex Cover

### ■ Vertex cover (VC)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E, u \in S$  or  $v \in S$  (or both); 0 otherwise

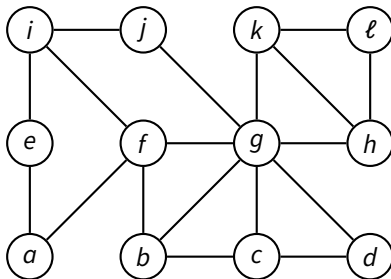


$K = 7$

## Example: Vertex Cover

### ■ Vertex cover (VC)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E, u \in S$  or  $v \in S$  (or both); 0 otherwise



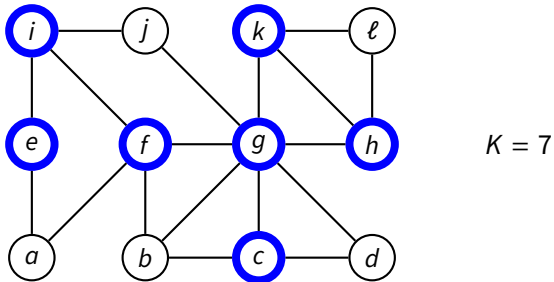
$K = 7$

### ■ VC $\in$ NP?

## Example: Vertex Cover

### ■ Vertex cover (VC)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise

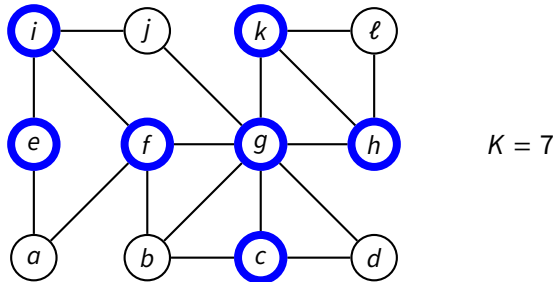


- $VC \in NP?$  Yes: given a vertex cover  $S$ , it is easy (poly-time) to verify that  $S$  is valid

## Example: Vertex Cover

### ■ Vertex cover (VC)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise



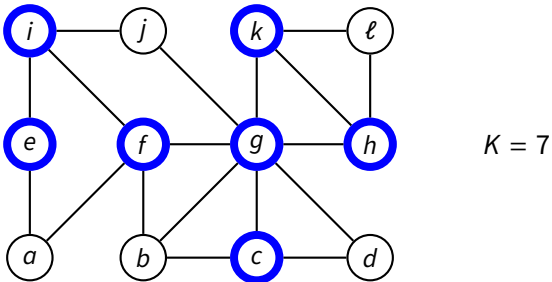
■  $VC \in NP?$  Yes: given a vertex cover  $S$ , it is easy (poly-time) to verify that  $S$  is valid

■  $VC \in P?$

## Example: Vertex Cover

### ■ Vertex cover (VC)

- ▶ *Input:* A graph  $G = (V, E)$  and a number  $K$
- ▶ *Output:* 1, if there is set  $S$  of at most  $k$  vertices such that for every edge  $e = (u, v) \in E$ ,  $u \in S$  or  $v \in S$  (or both); 0 otherwise

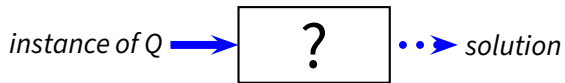


- $VC \in NP?$  Yes: given a vertex cover  $S$ , it is easy (poly-time) to verify that  $S$  is valid
- $VC \in P?$  We don't know

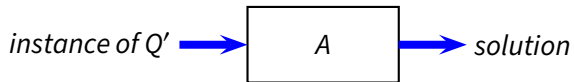
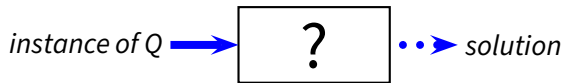


- In our theory of complexity, we want to argue that problem  $Q'$  is *just as hard* as problem  $Q$

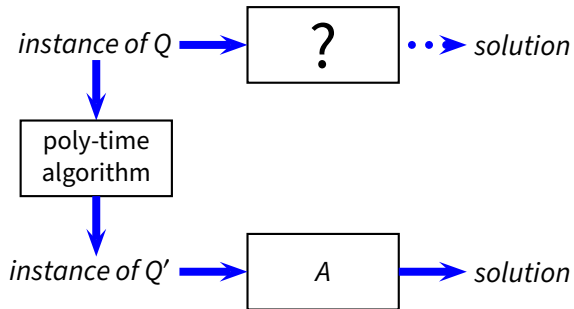
- In our theory of complexity, we want to argue that problem  $Q'$  is *just as hard* as problem  $Q$
- We do that with ***polynomial-time reductions***



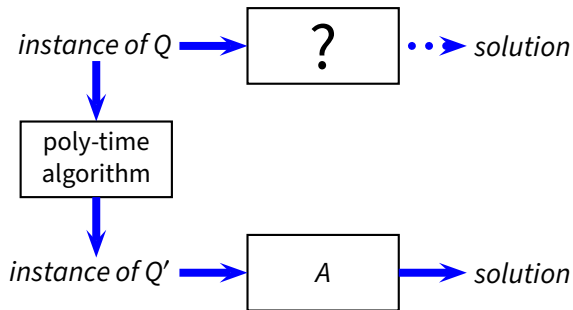
- In our theory of complexity, we want to argue that problem  $Q'$  is *just as hard* as problem  $Q$
- We do that with **polynomial-time reductions**



- In our theory of complexity, we want to argue that problem  $Q'$  is *just as hard* as problem  $Q$
- We do that with **polynomial-time reductions**

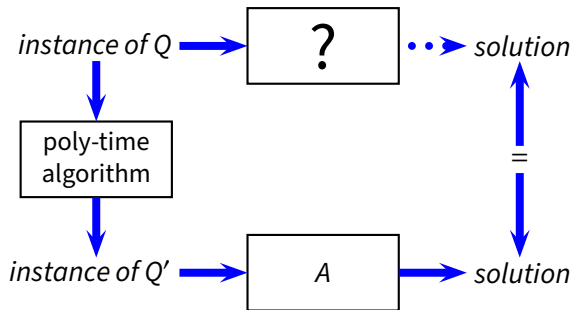


- In our theory of complexity, we want to argue that problem  $Q'$  is *just as hard* as problem  $Q$
- We do that with **polynomial-time reductions**



- ▶ an instance  $q$  of  $Q$  is transformed into an instance  $q'$  of  $Q'$  through a polynomial-time algorithm

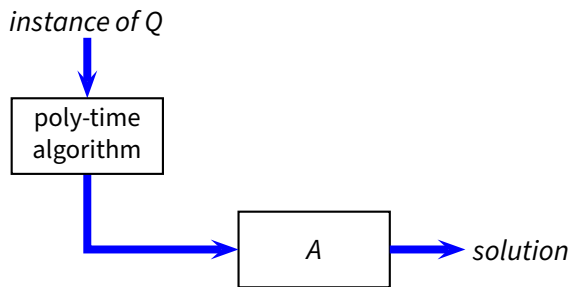
- In our theory of complexity, we want to argue that problem  $Q'$  is *just as hard* as problem  $Q$
- We do that with **polynomial-time reductions**



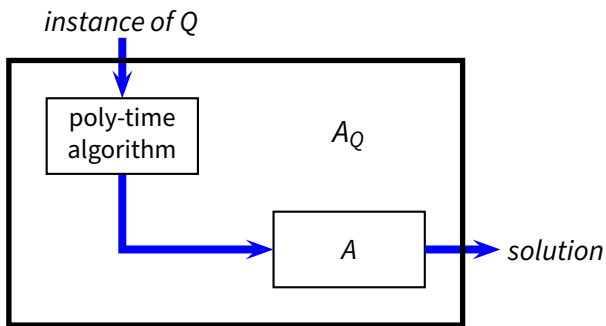
- ▶ an instance  $q$  of  $Q$  is transformed into an instance  $q'$  of  $Q'$  through a polynomial-time algorithm
- ▶ the solution to  $q$  is 1 if and only if the solution to  $q'$  is 1



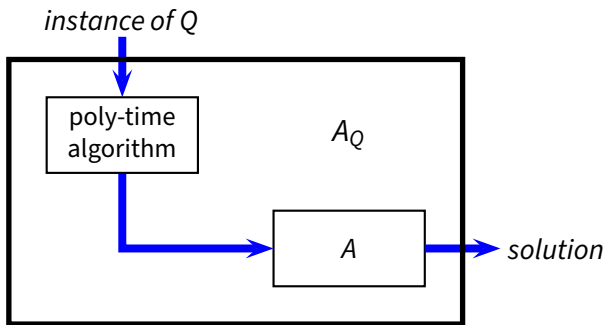
- Solution by polynomial-time reductions to a solvable problem



- Solution by polynomial-time reductions to a solvable problem

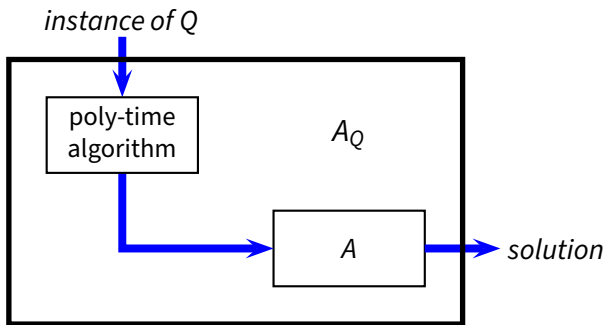


- Solution by polynomial-time reductions to a solvable problem



- ▶ if  $A$  is polynomial-time, then  $A_Q$  is also polynomial time

- Solution by polynomial-time reductions to a solvable problem



- ▶ if  $A$  is polynomial-time, then  $A_Q$  is also polynomial time
- ▶ therefore if  $Q' \in P$ , then  $Q \in P$



## ■ 2-CNF-SAT problem

### Input:

- ▶  $f$  is a Boolean formula of  $n$  (Boolean) variables  $x_1, x_2, \dots, x_n$
- ▶  $f$  is in *conjunctive normal form (CNF)*, so  $f = C_1 \wedge C_2 \wedge \dots \wedge C_k$
- ▶ every *clause*  $C_i$  of  $f$  contains exactly *two* literals (a variable or its negation)

### Output: 1 iff $f$ is satisfiable

- ▶ there is an assignment of variables that satisfies  $f$

### Example:

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

## 2-CNF-SAT to Implicative Form

## 2-CNF-SAT to Implicative Form

- Consider each clause  $C_i$

$$(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$$

so we can rewrite a 2-CNF-SAT formula  $f$  into another formula in *implicative normal form*

- **Example:**

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

## 2-CNF-SAT to Implicative Form

- Consider each clause  $C_i$

$$(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$$

so we can rewrite a 2-CNF-SAT formula  $f$  into another formula in *implicative normal form*

- **Example:**

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

is equivalent to

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2)$$

## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge$$

$$(x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$

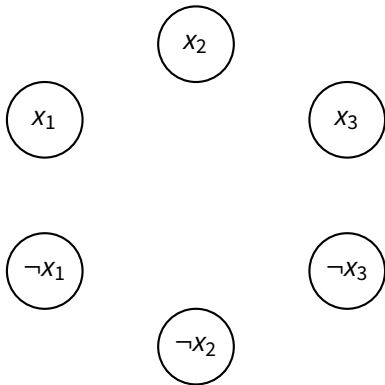
## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge$$

$$(x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$

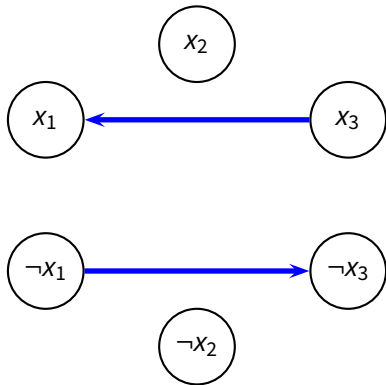


## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge \\ (x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$

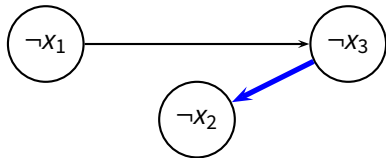
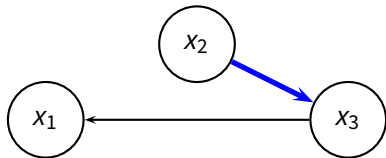


## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

↓↑

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2)$$
$$(x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$



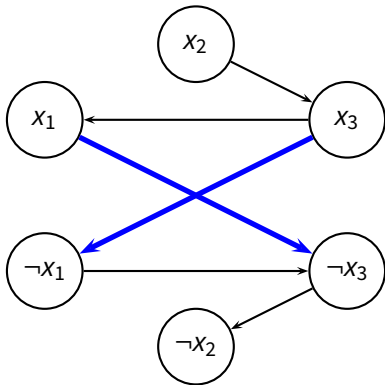
## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge$$

$$(x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$

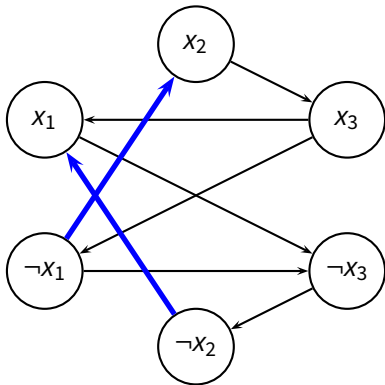


## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge$$
$$(x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$

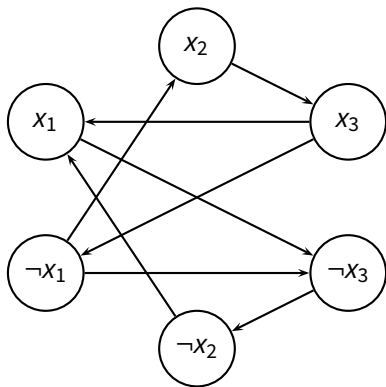


## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge \\ (x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$



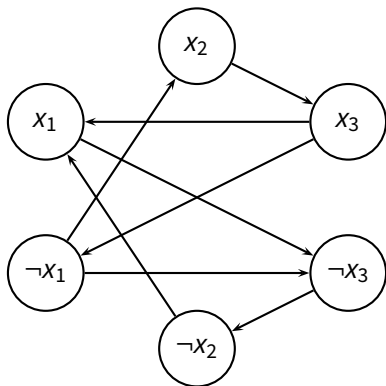
*not satisfiable*  
if and only if  
 $x_i \rightsquigarrow \neg x_i \rightsquigarrow x_i$   
for some  $i$

## 2-CNF-SAT to Graph Reachability

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$\Downarrow \Uparrow$

$$(\neg x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow x_1) \wedge (x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow \neg x_2) \wedge \\ (x_1 \Rightarrow \neg x_3) \wedge (x_3 \Rightarrow \neg x_1) \wedge (\neg x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow x_1)$$



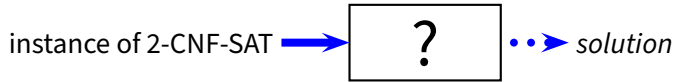
*not satisfiable*  
if and only if  
 $x_i \rightsquigarrow \neg x_i \rightsquigarrow x_i$   
for some  $i$

**depth-first search**

# Reduction of 2-CNF-SAT

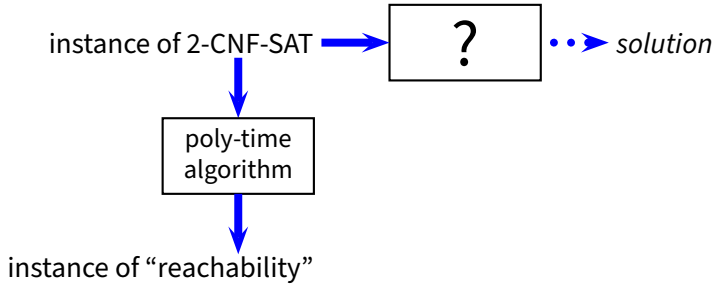
# Reduction of 2-CNF-SAT

- 2-CNF-SAT  $\in P$



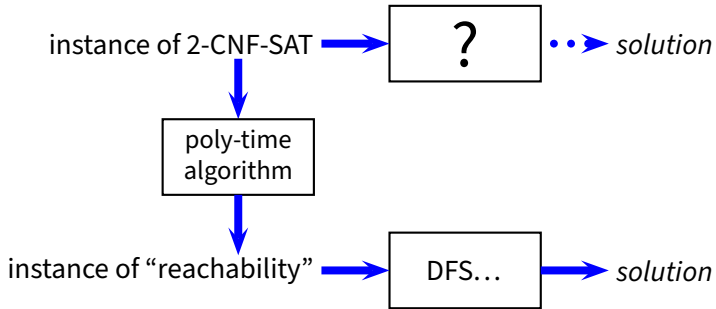
# Reduction of 2-CNF-SAT

- 2-CNF-SAT  $\in P$



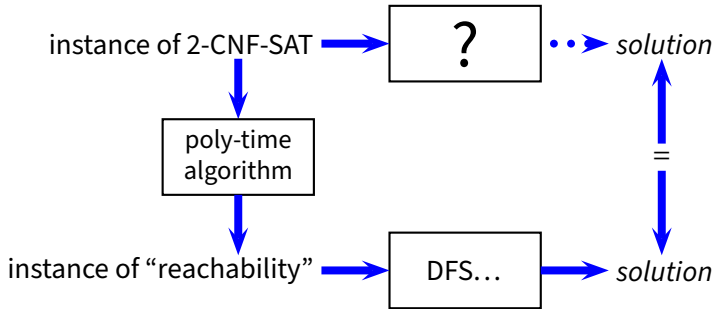
# Reduction of 2-CNF-SAT

- 2-CNF-SAT  $\in P$



# Reduction of 2-CNF-SAT

- 2-CNF-SAT  $\in P$





- A problem  $Q$  is ***polynomial-time reducible*** to another problem  $Q'$  if there is a *polynomial-time reduction*

- A problem  $Q$  is **polynomial-time reducible** to another problem  $Q'$  if there is a *polynomial-time reduction*
  - ▶ a polynomial-time algorithm transforms every instance  $q$  of  $Q$  into an instance  $q'$  of  $Q'$
  - ▶ the solution to  $q$  is 1 if and only if the solution to  $q'$  is 1

- A problem  $Q$  is **polynomial-time reducible** to another problem  $Q'$  if there is a *polynomial-time reduction*
  - ▶ a polynomial-time algorithm transforms every instance  $q$  of  $Q$  into an instance  $q'$  of  $Q'$
  - ▶ the solution to  $q$  is 1 if and only if the solution to  $q'$  is 1
- A problem  $Q'$  is **NP-hard** if *all problems  $Q \in NP$  are polynomial-time reducible to  $Q'$*

- A problem  $Q$  is **polynomial-time reducible** to another problem  $Q'$  if there is a *polynomial-time reduction*
  - ▶ a polynomial-time algorithm transforms every instance  $q$  of  $Q$  into an instance  $q'$  of  $Q'$
  - ▶ the solution to  $q$  is 1 if and only if the solution to  $q'$  is 1
- A problem  $Q'$  is **NP-hard** if *all problems*  $Q \in NP$  are *polynomial-time reducible* to  $Q'$
- A problem  $Q'$  is **NP-complete** if  $Q' \in NP$  and  $Q'$  is NP-hard

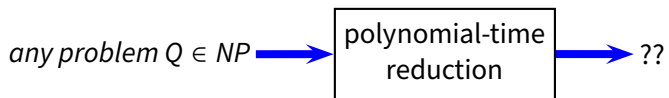
- A problem  $Q$  is **polynomial-time reducible** to another problem  $Q'$  if there is a *polynomial-time reduction*
  - ▶ a polynomial-time algorithm transforms every instance  $q$  of  $Q$  into an instance  $q'$  of  $Q'$
  - ▶ the solution to  $q$  is 1 if and only if the solution to  $q'$  is 1
- A problem  $Q'$  is **NP-hard** if *all problems*  $Q \in NP$  are *polynomial-time reducible* to  $Q'$
- A problem  $Q'$  is **NP-complete** if  $Q' \in NP$  and  $Q'$  is NP-hard
- If  $Q'$  is NP-hard and *polynomial-time reducible* to  $Q''$ , then  $Q''$  is NP-hard

- A problem  $Q$  is **polynomial-time reducible** to another problem  $Q'$  if there is a *polynomial-time reduction*
  - ▶ a polynomial-time algorithm transforms every instance  $q$  of  $Q$  into an instance  $q'$  of  $Q'$
  - ▶ the solution to  $q$  is 1 if and only if the solution to  $q'$  is 1
- A problem  $Q'$  is **NP-hard** if *all problems*  $Q \in NP$  are *polynomial-time reducible to*  $Q'$
- A problem  $Q'$  is **NP-complete** if  $Q' \in NP$  and  $Q'$  is NP-hard
- If  $Q'$  is NP-hard and *polynomial-time reducible to*  $Q''$ , then  $Q''$  is NP-hard
- If  $Q'$  is NP-hard and *polynomial-time solvable*, then  $P = NP$ 
  - ▶ most researchers believe that there is no such  $Q'$

# The First NP-Complete Problem

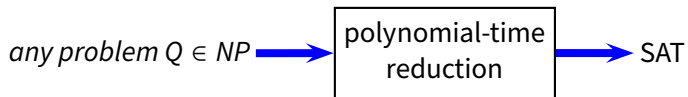
# The First NP-Complete Problem

- Is there any NP-complete problem?



# The First NP-Complete Problem

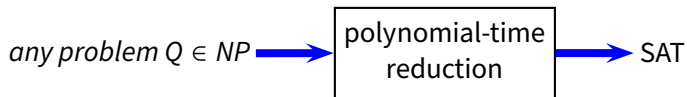
- Is there any NP-complete problem?



- *Circuit satisfiability (SAT)* was the first problem that was proved NP-hard and, since  $SAT \in NP$ , also NP-complete

# The First NP-Complete Problem

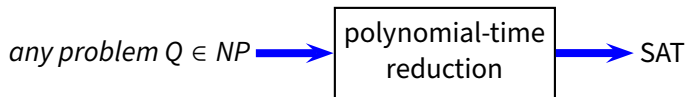
- Is there any NP-complete problem?



- *Circuit satisfiability (SAT)* was the first problem that was proved NP-hard and, since  $SAT \in NP$ , also NP-complete
- Many other problems were then proved NP-complete through polynomial reductions
  - ▶ e.g., SAT is polynomial-time reducible to Vertex Cover (and VC is in NP)
  - ▶ therefore, Vertex Cover is also NP-complete

# The First NP-Complete Problem

- Is there any NP-complete problem?



- *Circuit satisfiability (SAT)* was the first problem that was proved NP-hard and, since  $SAT \in NP$ , also NP-complete
- Many other problems were then proved NP-complete through polynomial reductions
  - ▶ e.g., SAT is polynomial-time reducible to Vertex Cover (and VC is in NP)
  - ▶ therefore, Vertex Cover is also NP-complete
- If a problem is NP-Hard (or NP-Complete) you should not feel so bad for not finding an efficient solution algorithm