



CS215 DISCRETE MATH

reduction 归约

correctness
complexity

Dr. QI WANG

Department of Computer Science and Engineering

Office: Room413, CoE South Tower

Email: wangqi@sustech.edu.cn

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have **small Big- O** running time.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have **small Big- O** running time.
- Being able to do good algorithm design lets you identify the **hard parts** of your problem and deal with them **effectively**.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have **small Big-O running time**.
- Being able to do good algorithm design lets you identify the **hard parts** of your problem and deal with them **effectively**.
- Too often, programmers try to solve problems using **brute force techniques** and end up with **slow complicated code!**

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have **small Big-O running time**.
- Being able to do good algorithm design lets you identify the **hard parts** of your problem and deal with them **effectively**.
- Too often, programmers try to solve problems using **brute force techniques** and end up with **slow complicated code!**
- A few hours of abstract thought devoted to algorithm design could have **speeded up the solution substantially and simplified it!**

Dealing with Hard Problems

- What happens if you **can't** find an efficient algorithm for a given problem?

Dealing with Hard Problems

- What happens if you **can't** find an efficient algorithm for a given problem?

Blame yourself.



I couldn't find a polynomial-time algorithm.
I guess I am too dumb.

Dealing with Hard Problems

- What happens if you **can't** find an efficient algorithm for a given problem?

Show that **no**-efficient algorithm exists.



I couldn't find a polynomial-time algorithm, because **no** such algorithm exists.

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:
“All” that is needed is to demonstrate an algorithm.

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:
“All” that is needed is to demonstrate an algorithm.
- Proving that no efficient algorithm exists for a particular problem is difficult:

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:
“All” that is needed is to demonstrate an algorithm.
- Proving that no efficient algorithm exists for a particular problem is difficult:

How can we prove the non-existence of something?

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

“All” that is needed is to demonstrate an algorithm.

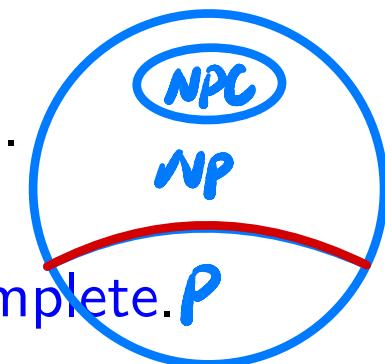
- Proving that no efficient algorithm exists for a particular problem is difficult:

How can we prove the non-existence of something?

We will now learn about NP-Complete problems, which provide us with a way to approach this question.

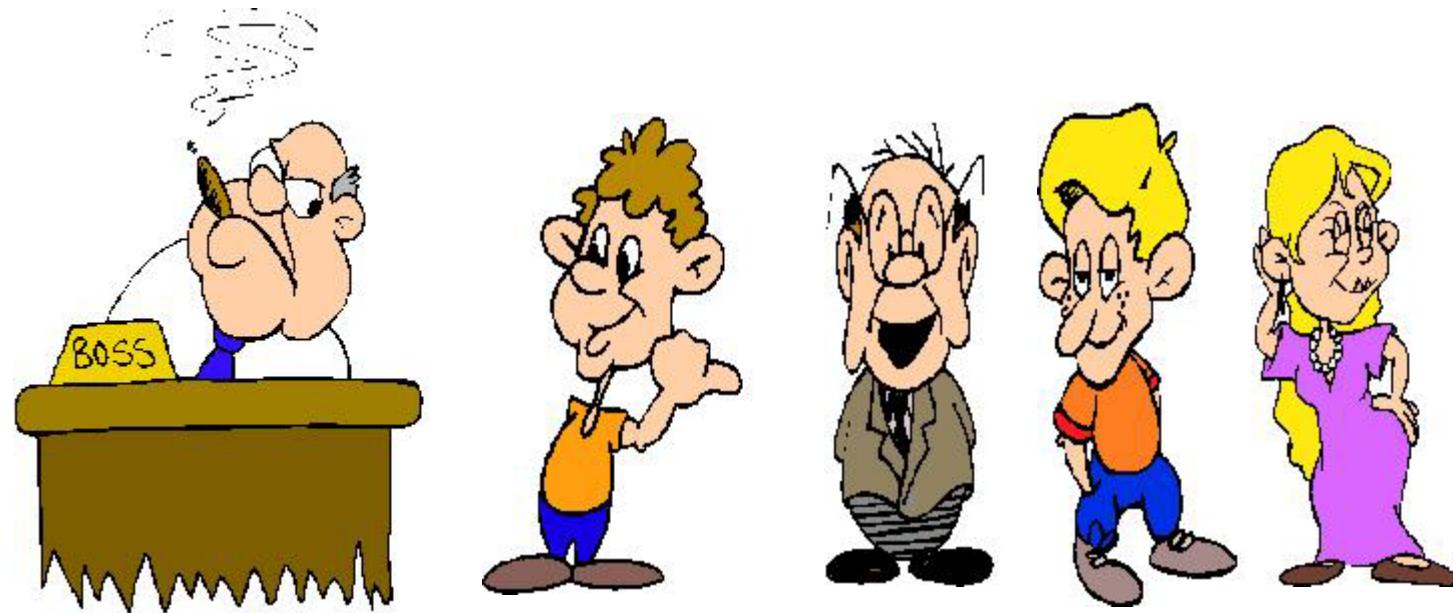
Introduction

- A very large class of thousands of practical problems for which it is **not** known if the problems have “**efficient**” solutions.
- It is known that if **any one** of the **NP-Complete** problems has an efficient solution then **all** of the NP-Complete problems have efficient solutions.
难度上大体一致
- Researchers have spent innumerable man-years trying to find efficient solutions to these problems but **failed**.
- So, **NP-Complete** problems are very likely to be **hard**.
- What do you do: prove that **your problem is NP-Complete**.



Introduction

What do you actually do:



I couldn't find a polynomial-time algorithm,
but neither could all these other smart people!

Encoding the Inputs of Problems

- Complexity of a problem is measure w.r.t the size of input.

Encoding the Inputs of Problems

- Complexity of a problem is measure w.r.t the size of input.
就是 input 的 **大小**
- In order to formally discuss how hard a problem is, we need to be **much more** formal than before about the **input size** of a problem.

The Input Size of Problems

- The **input size** of a problem might be defined in a number of ways.

The Input Size of Problems

- The **input size** of a problem might be defined in a number of ways.

Definition The *input size* of a problem is the **minimum number** of bits ($\{0,1\}$) needed to encode the input of the problem.

The Input Size of Problems

- The **input size** of a problem might be defined in a number of ways.

Definition The *input size* of a problem is the **minimum number** of bits ($\{0,1\}$) needed to **encode** the input of the problem.

- The **exact** input size s , determined by an **optimal** encoding method, is **hard** to compute in most cases.

The Input Size of Problems

- The **input size** of a problem might be defined in a number of ways.

Definition The *input size* of a problem is the **minimum number** of bits ($\{0,1\}$) needed to **encode** the input of the problem.

- The **exact** input size s , determined by an **optimal** encoding method, is **hard** to compute in most cases.

However, we do **not** need to determine s **exactly**.

For most problems, it is sufficient to choose some **natural**, and (usually) **simple**, encoding and use the size s of this encoding.

Input Size Example: Composite

■ Example:

Given a positive integer n , are there integers $j, k > 1$ such that $n = jk$? (i.e., **is n a composite number?**)

Input Size Example: Composite

■ Example:

Given a positive integer n , are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Question:

What is the input size of this problem?

Input Size Example: Composite

■ Example:

Given a positive integer n , are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Question:

What is the input size of this problem?

整数 n 用 $\{0,1\}$ 比特数表示

Any integer $n > 0$ can be represented in the binary number system as a string $a_0a_1 \cdots a_k$ of length $\lceil \log_2(n + 1) \rceil$.

Thus, a natural measure of input size is $\lceil \log_2(n + 1) \rceil$ (or just $\log_2 n$)

Input Size Example: Sorting

■ Example:

Sort n integers a_1, \dots, a_n

Input Size Example: Sorting

■ Example:

Sort n integers a_1, \dots, a_n

Question:

What is the input size of this problem?

$$\begin{aligned}x &= \log a_1 + \log a_2 + \dots + \log a_n \\&\leq n \log \max \{a_1, a_2, \dots, a_n\} \\&\leq n C \quad * \text{时间复杂度表示成 } n \\&\quad \text{与表示成 } x \text{ 是一致的 (更严格)}\end{aligned}$$

Input Size Example: Sorting

■ Example:

Sort n integers a_1, \dots, a_n

Question:

What is the input size of this problem?

Using fixed length encoding, we write a_i as a binary string of length $m = \lceil \log_2 \max(|a_i| + 1) \rceil$.

This coding gives an input size nm .

Complexity in terms of Input Size

■ Example: (Composite)

The naive algorithm for determining whether n is composite compares n with the first $n - 1$ numbers to see if **any** of them divides n

$$n \div 2 = q \cdots r \quad r \neq 0$$

$$\begin{array}{r} n \div 3 \\ \vdots \end{array}$$

$\circlearrowleft \sqrt{n}$

$$n \div (n-1) \Rightarrow O(n) \text{ divisions}$$

$$x = \log_2 n$$

$$f = O(n) = O(2^x) \text{ 就这个算法而言是指数的}$$

Complexity in terms of Input Size

■ Example: (Composite)

The naive algorithm for determining whether n is composite compares n with the first $n - 1$ numbers to see if **any** of them divides n

This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

Complexity in terms of Input Size

■ Example: (Composite)

The naive algorithm for determining whether n is composite compares n with the first $n - 1$ numbers to see if **any** of them divides n

This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

But, note that the input size of this problem is $\text{size}(n) = \log_2 n$, so the number of comparisons performed is actually $\Theta(n) = \Theta(2^{\text{size}(n)})$, which is exponential.

Input Size of Problems

- **Definition** Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n , where $a_1, b_1, c_1, a_2, b_2, c_2$ are **some** positive constants.

Input Size of Problems

- **Definition** Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n , where $a_1, b_1, c_1, a_2, b_2, c_2$ are **some** positive constants.

Example:

All polynomials are of the **same type**, but *polynomials* and *exponentials* are of **different types**.

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

Compute $a \times b$.

$$\log_2 \max\{a, b\} \leq \log_2 a + \log_2 b \leq 2 \log_2 \max\{a, b\}$$

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

Compute $a \times b$.

Question:

What is the input size of this problem?

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

Compute $a \times b$.

Question:

What is the input size of this problem?

The minimum input size is

$$s = \lceil \log_2(a + 1) \rceil + \lceil \log_2(b + 1) \rceil.$$

A natural choice is to use $t = \log_2 \max(a, b)$ since $\frac{s}{2} \leq t \leq s$.

Decision Problems

判定性問題

- **Definition** A *decision problem* is a question that has two possible answers: *yes* and *no*.

Decision Problems

- **Definition** A *decision problem* is a question that has two possible answers: yes and no.

If L is the problem, and x is the input, we will often write $x \in L$ to denote a yes answer and $x \notin L$ to denote a no answer.

Optimization Problems

- **Definition** An *optimization problem* requires an answer that is an optimal configuration.

Optimization Problems

- **Definition** An *optimization problem* requires an answer that is an optimal configuration.

An *optimization problem* usually has a corresponding decision problem.

Optimization Problems

- **Definition** An *optimization problem* requires an answer that is an optimal configuration.

An *optimization problem* usually has a corresponding decision problem.

Examples:

Knapsack vs. Decision Knapsack (DKnapsack)

Knapsack vs. DKnapsack

- We have a knapsack of capacity W (a positive integer) and n objects with weights w_1, \dots, w_n and values v_1, \dots, v_n , where v_i and w_i are positive integers.

Knapsack vs. DKnapsack

- We have a knapsack of capacity W (a positive integer) and n objects with weights w_1, \dots, w_n and values v_1, \dots, v_n , where v_i and w_i are positive integers.

Optimization problem: (Knapsack) ~~更难一些~~

Find the largest value $\sum_{i \in T} v_i$ of any subset T that fits in the knapsack, i.e., $\sum_{i \in T} w_i \leq W$.

Decision problem: (DKnapsack)

Given k , is there a subset of the objects that fits in the knapsack and has total value at least k ?

Optimization and Decision Problems

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

Optimization and Decision Problems

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

First solve the optimization problem, then check the decision problem. If it does, answer yes, otherwise no.

Optimization and Decision Problems

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

First solve the optimization problem, then check the decision problem. If it does, answer yes, otherwise no.

Thus, if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

Complexity Classes

- The Theory of Complexity deals with
 - ◊ the classification of certain “decision problems” into several classes:
 - ◊ the class of “easy” problems
 - ◊ the class of “hard” problems
 - ◊ the class of “hardest” problems

Complexity Classes

- The Theory of Complexity deals with
 - ◊ the classification of certain “decision problems” into several classes:
 - ◊ the class of “easy” problems
 - ◊ the class of “hard” problems
 - ◊ the class of “hardest” problems
 - ◊ relations among the three classes

Complexity Classes

- The Theory of Complexity deals with
 - ◊ the classification of certain “decision problems” into several classes:
 - ◊ the class of “easy” problems
 - ◊ the class of “hard” problems
 - ◊ the class of “hardest” problems
 - ◊ relations among the three classes
 - ◊ properties of problems in the three classes

Complexity Classes

- The Theory of Complexity deals with
 - ◊ the classification of certain “decision problems” into several classes:
 - ◊ the class of “easy” problems
 - ◊ the class of “hard” problems
 - ◊ the class of “hardest” problems
 - ◊ relations among the three classes
 - ◊ properties of problems in the three classes

Question:

How to classify decision problems?

Complexity Classes

- The Theory of Complexity deals with
 - ◊ the classification of certain “decision problems” into several classes:
 - ◊ the class of “easy” problems
 - ◊ the class of “hard” problems
 - ◊ the class of “hardest” problems
 - ◊ relations among the three classes
 - ◊ properties of problems in the three classes

Question:

How to classify decision problems?

A. Use polynomial-time algorithms.

Polynomial-Time Algorithms

- **Definition** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n , and n is the **input size** of the problem that the algorithm solves.

Polynomial-Time Algorithms

- **Definition** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n , and n is the **input size** of the problem that the algorithm solves.

Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will **not** affect the conclusion of whether an algorithm is **polynomial-time**.

Polynomial-Time Algorithms

- **Definition** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n , and n is the **input size** of the problem that the algorithm solves.

Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will **not** affect the conclusion of whether an algorithm is polynomial-time.

Example:

The standard multiplication algorithm has time $O(m_1 m_2)$, where m_1, m_2 denote the number of digits in the two integers, respectively.

判度函数 $f = O(z^x x^2)$

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is **not** $O(n^k)$ for any fixed $k \geq 0$.

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is **not** $O(n^k)$ for any fixed $k \geq 0$.

Let's return to the [Composite](#) problem.

- ◊ it checks, in time $\Theta((\log n)^2)$, whether k divides n for each k with $2 \leq k \leq n - 1$.
- ◊ The complete algorithm therefore uses $\Theta(n(\log n)^2)$ time.

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is **not** $O(n^k)$ for any fixed $k \geq 0$.

Let's return to the [Composite](#) problem.

- ◊ it checks, in time $\Theta((\log n)^2)$, whether k divides n for each k with $2 \leq k \leq n - 1$.
- ◊ The complete algorithm therefore uses $\Theta(n(\log n)^2)$ time.

Conclusion: The algorithm is **nonpolynomial!**

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is **not** $O(n^k)$ for any fixed $k \geq 0$.

Let's return to the [Composite](#) problem.

- ◊ it checks, in time $\Theta((\log n)^2)$, whether k divides n for each k with $2 \leq k \leq n - 1$.
- ◊ The complete algorithm therefore uses $\Theta(n(\log n)^2)$ time.

Conclusion: The algorithm is **nonpolynomial!**

Question:

Why?

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is **not** $O(n^k)$ for any fixed $k \geq 0$.

Let's return to the **Composite** problem.

- ◊ it checks, in time $\Theta((\log n)^2)$, whether k divides n for each k with $2 \leq k \leq n - 1$.
- ◊ The complete algorithm therefore uses $\Theta(n(\log n)^2)$ time.

Conclusion: The algorithm is **nonpolynomial!**

Question:

Why?

In terms of the **input size**, the complexity is $\Theta(2^N N^2)$.

Polynomial- vs. Nonpolynomial-Time

- Nonpolynomial-time algorithms are **impractical**.

2^n for $n = 100$: it takes **billions** of years!!!

Polynomial- vs. Nonpolynomial-Time

- Nonpolynomial-time algorithms are **impractical**.

2^n for $n = 100$: it takes **billions** of years!!!

In reality, an $O(n^{20})$ algorithm is **not** really practical.

Polynomial-Time Solvable Problems

- **Definition** A problem is *solvable in polynomial time* (or more simply, the problem is *in polynomial time*) if there exists an algorithm which solves the problem in polynomial time (a.k.a. *tractable*).
容易解决的

Polynomial-Time Solvable Problems

- **Definition** A problem is *solvable in polynomial time* (or more simply, the problem is *in polynomial time*) if there exists an algorithm which solves the problem in polynomial time (a.k.a. *tractable*).

Definition (The Class P) The class P consists of all **decision problems** that are solvable in **polynomial time**. That is, there exists an algorithm that will decide in **polynomial time** if any given input is a **yes-input** or a **no-input**.

The Class P

■ Question:

How to prove that a decision problem is in P?

The Class P

- **Question:**

How to prove that a decision problem is in P?

A. Find a **polynomial-time** algorithm.

The Class P

- **Question:**

How to prove that a decision problem is in P?

A. Find a **polynomial-time** algorithm.

- Question:**

How to prove that a decision problem is **not** in P?

The Class P

- **Question:**

How to prove that a decision problem is in P?

A. Find a **polynomial-time** algorithm.

Question:

How to prove that a decision problem is **not** in P?

A. You need to prove that there is **no** polynomial-time algorithm for this problem. (**much much harder**)

Certificates and Verifying Certificates

- **Observation:** A decision problem is usually formulated as:
Is there an object satisfying some conditions?

Certificates and Verifying Certificates

- **Observation:** A decision problem is usually formulated as:
Is there an object satisfying some conditions?



Certificates and Verifying Certificates

- A *certificate* is a specific object corresponding to a *yes-input*, such that it can be used to show that the input is **indeed** a yes-input.

Certificates and Verifying Certificates

- A *certificate* is a specific object corresponding to a *yes-input*, such that it can be used to show that the input is **indeed** a yes-input.

Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a *yes-input*.

The Class NP

- **Definition** The class NP consists of all decision problems such that, for each *yes-input*, there exists a *certificate* which allows one to verify in *polynomial time* that the input is indeed a *yes-input*. (看例题)

The Class NP

- **Definition** The class NP consists of all decision problems such that, for each yes-input, there exists a *certificate* which allows one to verify in polynomial time that the input is indeed a yes-input.

非确定的

NP – “nondeterministic polynomial-time”

P – DFA 确定有限状态自动机

NP – NFA 非 ~

Composite \in NP

- For Composite, a yes-input is just the integer n that is composite.

Composite \in NP

- For Composite, a **yes-input** is just the integer n that is composite.

Question: (Certificate)

What is needed to show n is actually a **yes-input**?

Composite \in NP

- For Composite, a **yes-input** is just the integer n that is composite.

Question: (Certificate)

What is needed to show n is actually a **yes-input**?

A. An integer a ($1 < a < n$) with the property that $a|n$.

Composite \in NP

- For Composite, a yes-input is just the integer n that is composite.

Question: (Certificate)

What is needed to show n is actually a yes-input?

A. An integer a ($1 < a < n$) with the property that $a|n$.

- Given a certificate a , check whether a divides n .
- This can be done in $O((\log n)^2)$. 做一次除法
- Composite \in NP

Composite \in NP

- For Composite, a yes-input is just the integer n that is composite.

Question: (Certificate)

What is needed to show n is actually a yes-input?

A. An integer a ($1 < a < n$) with the property that $a|n$.

- Given a certificate a , check whether a divides n .
- This can be done in $O((\log n)^2)$.
- Composite \in NP

DKnapsack \in NP

有一种装背包的方法 \Rightarrow

P = NP?

- One of the **most important** problems in CS is whether $P = NP$ or $P \neq NP$?

P = NP?

- One of the **most important** problems in CS is whether $P = NP$ or $P \neq NP$?
- Observe that $P \subseteq NP$.

P = NP?

- One of the **most important** problems in CS is whether $P = NP$ or $P \neq NP$?
- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is **doubtful**.

P = NP?

- One of the **most important** problems in CS is whether $P = NP$ or $P \neq NP$?
- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is **doubtful**.

Just being able to verify a certificate in **polynomial time** does **not** necessarily mean we can tell whether an input is a yes-input or a no-input in polynomial time.

P = NP?

- One of the **most important** problems in CS is whether $P = NP$ or $P \neq NP$?
- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is **doubtful**.

Just being able to verify a certificate in **polynomial time** does **not** necessarily mean we can tell whether an input is a yes-input or a no-input in **polynomial time**.

However, we are still **no** closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into **what distinguishes** an “easy” problem from a “hard” one.

What is Reduction?

- *Reduction* is a relationship between problems.

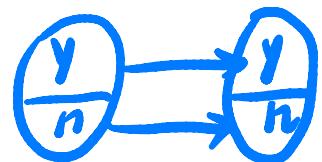
What is Reduction?

- *Reduction* is a relationship between problems.
- Problem Q can be *reduced* to Q' if every instance of Q can be “*rephrased*” to an instance of Q' .

判定版本可归约到优化版本

$$Q \leq Q'$$

input \rightsquigarrow input



What is Reduction?

- *Reduction* is a relationship between problems.
- **Problem Q** can be *reduced* to **Q'** if every instance of **Q** can be “*rephrased*” to an instance of **Q'**.
- **Example**

Q: multiplying two positive numbers

Q': adding two numbers

What is Reduction?

- *Reduction* is a relationship between problems.
- **Problem Q** can be *reduced* to **Q'** if every instance of **Q** can be “*rephrased*” to an instance of **Q'**.
- **Example**

Q: multiplying two positive numbers

Q': adding two numbers

Q can be *reduced* to **Q'** via a logarithmic transformation

$$xy = \exp[\log x + \log y]$$

What is Reduction?

- *Reduction* is a relationship between problems.
- **Problem Q** can be *reduced* to **Q'** if every instance of **Q** can be “*rephrased*” to an instance of **Q'**.
- **Example**

Q: multiplying two positive numbers

Q': adding two numbers

Q can be *reduced* to **Q'** via a logarithmic transformation

$$xy = \exp[\log x + \log y]$$

- If **Q** can be reduced to **Q'**,
then **Q** is “**no harder to solve**” than **Q'**.

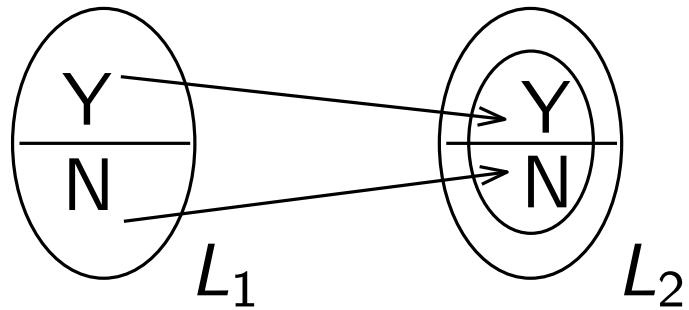


Polynomial-Time Reductions

- Let L_1 and L_2 be two decision problems

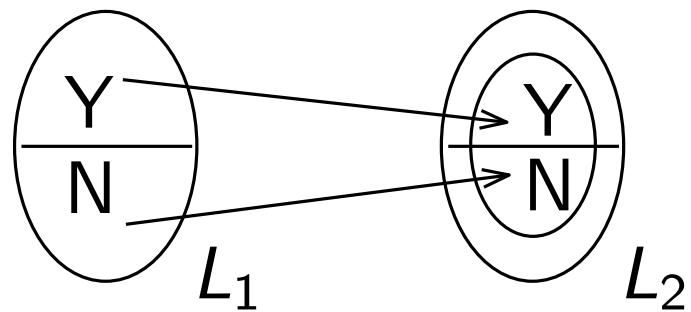
Polynomial-Time Reductions

- Let L_1 and L_2 be two decision problems
多项式时间归约
- A *polynomial-time reduction* from L_1 to L_2 is a transformation f with the following two properties:
 - (1) f transforms an input x for L_1 into an input $f(x)$ for L_2 s.t.
a yes-input of L_1 maps to a yes-input of L_2 , and a no-input of L_1 maps to a no-input of L_2
 - (2) f is computable in *polynomial time* in $\text{size}(x)$



Polynomial-Time Reductions

- Let L_1 and L_2 be two decision problems
- A *polynomial-time reduction* from L_1 to L_2 is a transformation f with the following two properties:
 - (1) f transforms an input x for L_1 into an input $f(x)$ for L_2 s.t.
a yes-input of L_1 maps to a yes-input of L_2 , and a no-input of L_1 maps to a no-input of L_2
 - (2) f is computable in *polynomial time* in $\text{size}(x)$



If such an f exists, we say that L_1 is *polynomial-time reducible* to L_2 , and write $L_1 \leq_P L_2$.

Polynomial-Time Reductions

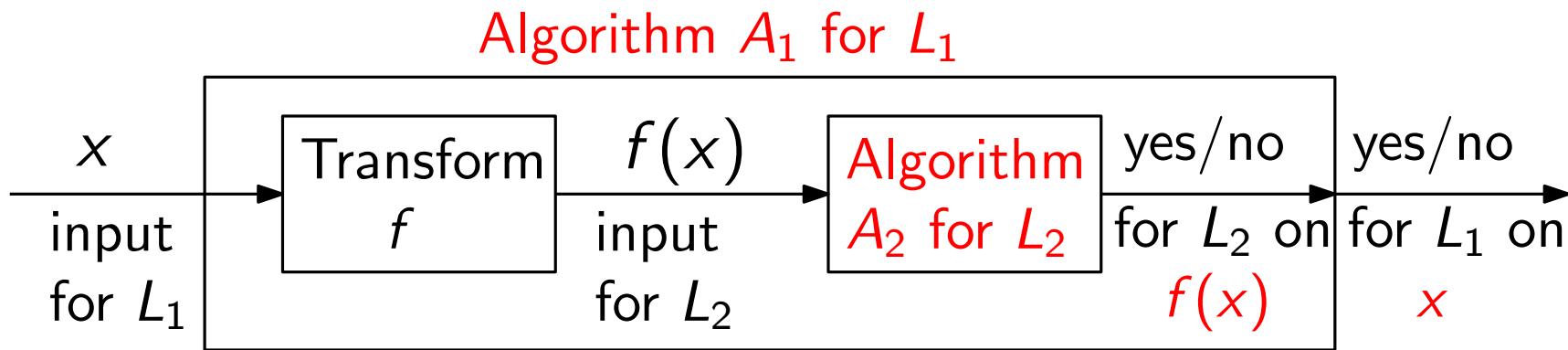
- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2

Polynomial-Time Reductions

- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2
如果能解决 L_2 , 就一定能解决 L_1
- Given an algorithm A_2 for the decision problem L_2 , we can develop an algorithm A_1 to solve L_1 :

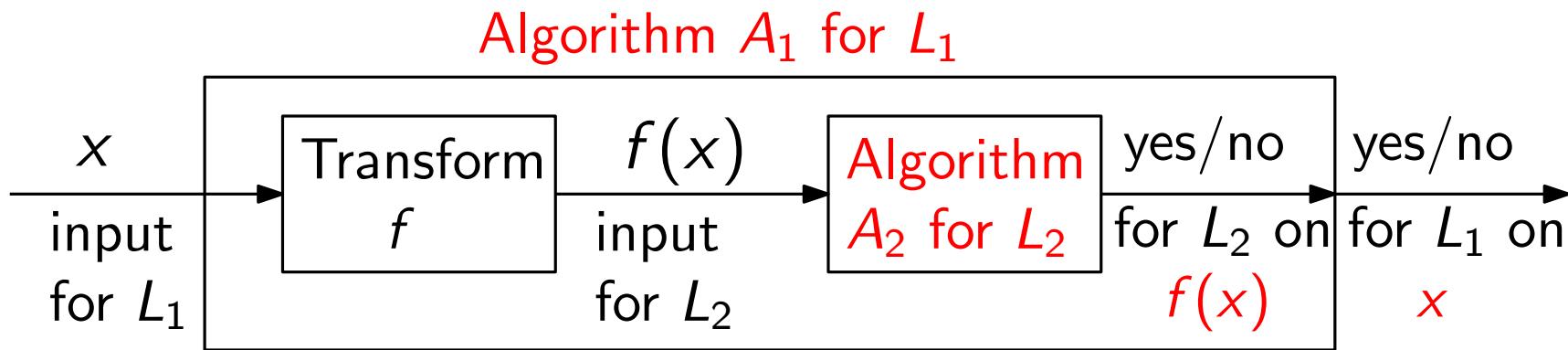
Polynomial-Time Reductions

- Intuitively, $L_1 \leq_P L_2$ means that L_1 is **no harder** than L_2
- Given an algorithm A_2 for the decision problem L_2 , we can develop an algorithm A_1 to solve L_1 :



Polynomial-Time Reductions

- Intuitively, $L_1 \leq_P L_2$ means that L_1 is **no harder** than L_2
- Given an algorithm A_2 for the decision problem L_2 , we can develop an algorithm A_1 to solve L_1 :



- If A_2 is polynomial-time algorithm, so is A_1

Polynomial-Time Reduction $f : L_1 \rightarrow L_2$

- **Theorem** If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$

Polynomial-Time Reduction $f : L_1 \rightarrow L_2$

- **Theorem** If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$

Proof. $L_2 \in P$ means we have a polynomial-time algorithm A_2 for L_2 . Since $L_1 \leq_P L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2 .

Polynomial-Time Reduction $f : L_1 \rightarrow L_2$

- **Theorem** If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$

Proof. $L_2 \in P$ means we have a polynomial-time algorithm A_2 for L_2 . Since $L_1 \leq_P L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2 .

Combining these, we get the following polynomial-time algorithm for solving L_1 :

- (1) take input x for L_1 and compute $f(x)$
- (2) run algorithm A_2 on input $f(x)$, and return the ans. (for L_2 on $f(x)$) as the ans. for L_1 on x

Polynomial-Time Reduction $f : L_1 \rightarrow L_2$

- **Theorem** If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$

Proof. $L_2 \in P$ means we have a polynomial-time algorithm A_2 for L_2 . Since $L_1 \leq_P L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2 .

Combining these, we get the following polynomial-time algorithm for solving L_1 :

- (1) take input x for L_1 and compute $f(x)$
- (2) run algorithm A_2 on input $f(x)$, and return the ans. (for L_2 on $f(x)$) as the ans. for L_1 on x

Both steps take polynomial time. So the combined algorithm takes polynomial time. Hence, $L_1 \in P$.

Polynomial-Time Reduction $f : L_1 \rightarrow L_2$

- **Theorem** If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$

Proof. $L_2 \in P$ means we have a polynomial-time algorithm A_2 for L_2 . Since $L_1 \leq_P L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2 .

Combining these, we get the following polynomial-time algorithm for solving L_1 :

- (1) take input x for L_1 and compute $f(x)$
- (2) run algorithm A_2 on input $f(x)$, and return the ans. (for L_2 on $f(x)$) as the ans. for L_1 on x

Both steps take polynomial time. So the combined algorithm takes polynomial time. Hence, $L_1 \in P$.

Note: The converse (if $L_1 \leq_P L_2$ and $L_1 \in P$, then $L_2 \in P$) is not true.

Transitivity of Reductions

- **Lemma** If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Transitivity of Reductions

- **Lemma** If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.
- Proof.**

Transitivity of Reductions

- **Lemma** If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Proof.

- The class *NP-Complete (NPC)*

The class *NPC* of NP-Complete problems consists of all decision problems L s.t.

- (1) $L \in NP$
- (2) for every $L' \in NP$, $L' \leq_P L$

Transitivity of Reductions

- **Lemma** If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Proof.

- The class *NP-Complete (NPC)*

The class *NPC* of NP-Complete problems consists of all decision problems L s.t.

- (1) $L \in NP$
- (2) for every $L' \in NP$, $L' \leq_P L$

Intuitively, *NPC* consists of all the **hardest** problems in NP.

NP-Completeness and Its Properties

- **Theorem** Let L be any problem in NPC.
 - (1) If **there is** a polynomial-time algorithm for L , then there is a polynomial-time algorithm for **every** $L' \in NP$
 - (2) If **there is no** polynomial-time algorithm for L , then there is no polynomial-time algorithm for **every** $L' \in NPC$

NP-Completeness and Its Properties

- **Theorem** Let L be any problem in NPC.

$$L' \leq_p L$$

- (1) If **there is** a polynomial-time algorithm for L , then there is a polynomial-time algorithm for **every** $L' \in NP$
- (2) If **there is no** polynomial-time algorithm for L , then there is no polynomial-time algorithm for **every** $L' \in NPC$

假设有

$$LENPC \subseteq NP$$

$$L \leq_p L'$$

- Either **all** NP-Complete problems are polynomial time solvable, or **all** NP-Complete problems are **not** polynomial time solvable.

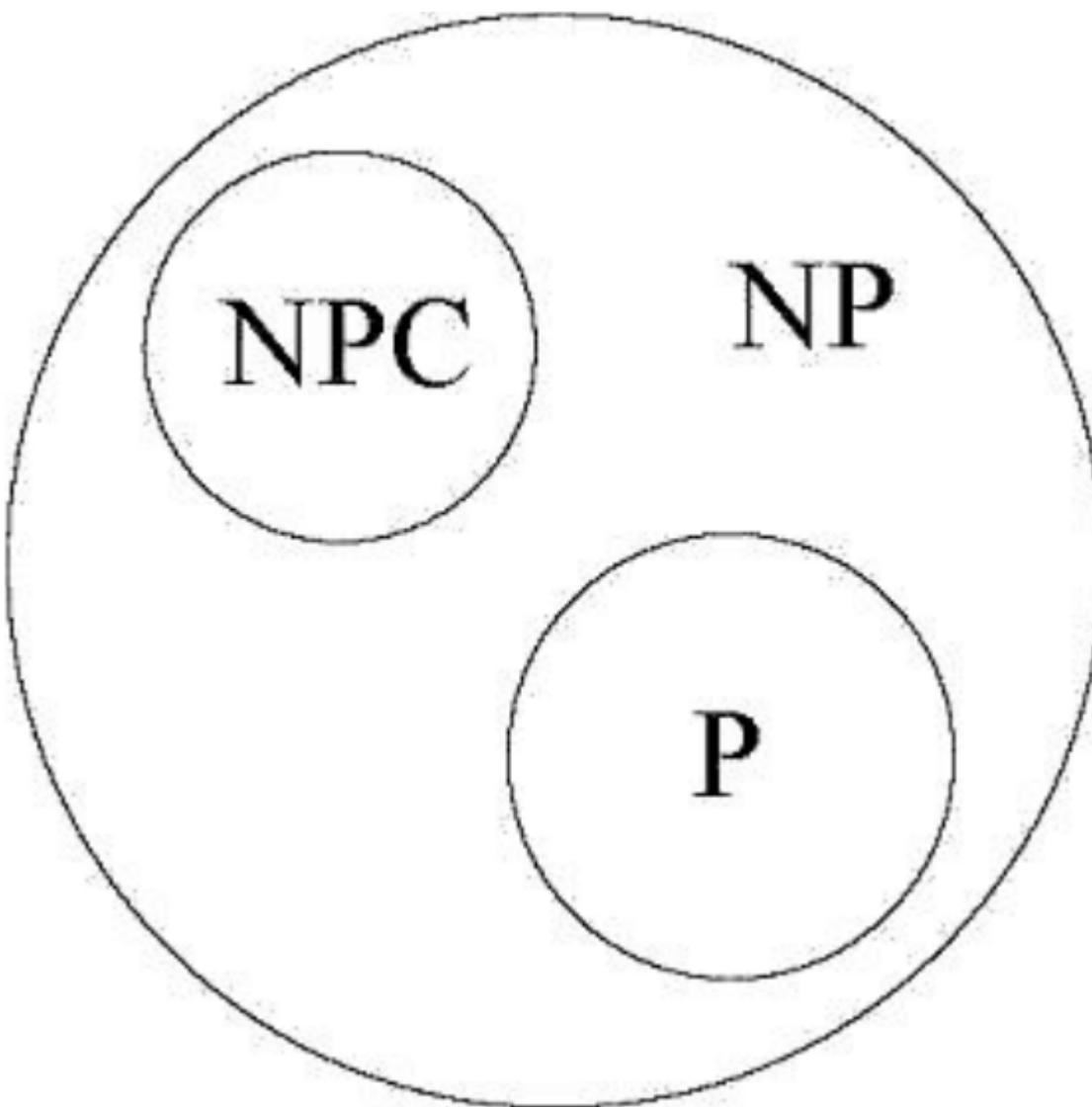
SAT问题与图论算法归纳

NP-Completeness and Its Properties

- **Theorem** Let L be any problem in NPC.
 - (1) If **there is** a polynomial-time algorithm for L , then there is a polynomial-time algorithm for **every** $L' \in NP$
 - (2) If **there is no** polynomial-time algorithm for L , then there is no polynomial-time algorithm for **every** $L' \in NPC$
- Either **all** NP-Complete problems are polynomial time solvable, or **all** NP-Complete problems are **not** polynomial time solvable.

This is the major reason why we are interested in NP-Completeness.

The Classes P, NP, and NPC



Next Lecture

- number theory ...

