

CSCI471/971

# Modern Cryptography

Computational Complexity & Cryptographic  
Notions

Jiageng Chen

Central China Normal University Wollongong Joint Institute

# *Modern* Cryptography

- Any cryptography can be broken but computationally hard
- How much hard?
- For example, if cryptography is well-designed, it will take at least 200 years using all computers in the whole world.
- What is computationally hard?

# The One Time Pad (Vernam 1917)

Very fast enc/dec !!

... but long keys (as long as plaintext)

Is the OTP secure? What is a secure cipher?

# What is a secure cipher?

Attacker's abilities: **CT only attack** (for now)

Possible security requirements:

attempt #1: **attacker cannot recover secret key**

$E(k, m) = m$  would be secure

attempt #2: **attacker cannot recover all of plaintext**

$E(k, m_0 || m_1) = m_0 || k \oplus m_1$  would be secure

Shannon's idea:

**CT should reveal no "info" about PT**

# Information Theoretic Security

## (Shannon 1949)

**Definition**      **(perfect security).** *Let  $\mathcal{E} = (E, D)$  be a Shannon cipher defined over  $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ . Consider a probabilistic experiment in which the random variable  $\mathbf{k}$  is uniformly distributed over  $\mathcal{K}$ . If for all  $m_0, m_1 \in \mathcal{M}$ , and all  $c \in \mathcal{C}$ , we have*

$$\Pr[E(\mathbf{k}, m_0) = c] = \Pr[E(\mathbf{k}, m_1) = c],$$

*then we say that  $\mathcal{E}$  is a **perfectly secure** Shannon cipher.*

# Information Theoretic Security

**Def:** A cipher  $(E,D)$  over  $(K,M,C)$  has **perfect secrecy** if

$$\forall m_0, m_1 \in M \quad ( |m_0| = |m_1| ) \quad \text{and} \quad \forall c \in C$$

$$Pr[ E(k,m_0)=c ] = Pr[ E(k,m_1)=c ] \quad \text{where } k \leftarrow K^R$$

- 
1. Given CT cannot tell if message is  $m_0$  or  $m_1$  (For all  $m_0$  and  $m_1$ )
  2. Most powerful adversary learns nothing about Plaintext from ciphertext
  3. No Ciphertext only attack

Lemma: OTP has perfect secrecy.

Proof:  $\forall m, c: \Pr[E(k, m) = c] = \frac{\# \text{ keys } k \in K \text{ s.t. } E(k, m) = c}{|K|}$

So: if  $\forall m, c: \exists \Pr[k \in K: E(k, m) = c] = \text{const.}$



Cipher has perfect secrecy.

Let  $m \in \mathcal{M}$  and  $c \in \mathcal{C}$  .

How many OTP keys map  **$m$**  to  **$c$**  ?



Lemma: OTP has perfect secrecy.

Proof:

For OTP:  $\forall m, c: \text{if } E(k, m) = c$

$$k \oplus m = c \quad \longrightarrow \quad k = c \oplus m$$

$$\exists \Pr[ k \in K : E(k, m) = c ] = 1/|K|$$

OTP has perfect secrecy.

The bad news ...

Thm: perfect secrecy  $\Rightarrow |\mathcal{K}| \geq |\mathcal{M}|$

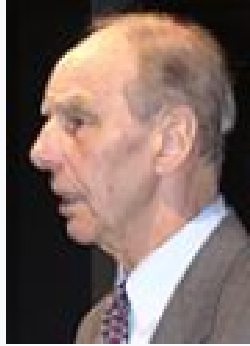

Perfect secrecy  Key length  $\geq$  message length

Hard to apply in practice!

# Computational Complexity Theory

We are going to use one week to learn what is computationally hard

# 1965: paper with **Turing Award** "On the Computational Complexity of Algorithms"

|      |                    |   |   |
|------|--------------------|---|---|
| 1993 | Juris Hartmanis    |   | In recognition of their seminal paper which established the foundations for the field of computational complexity theory. <sup>[32]</sup> |
|      | Richard E. Stearns |  |   |

# Concepts in Complexity

- Problem
  - Computers
  - Algorithm
- 
- Efficiency of Algorithm: Cost
  - Complexity of Algorithm: Speed

# Computing Problem

- Computing problem = A set of instances and solutions
- A computing problem is given an instance and to find its solution
- Instance length and instance number
- Two types:
  - Decisional Problem
  - Computational Problem (Search Problem)

# Decisional Problem

- A decisional problem is a computational problem where the answer for every instance is either yes or no.

An example of a decision problem is primality testing:

"Given a positive integer  $n$ , determine if  $n$  is prime."

- A decision problem is typically represented as the set of all instances for which the answer is **yes**. It also known as a **Language**.

For example, primality testing can be represented as the infinite set

$$L = \{2, 3, 5, 7, 11, \dots\}$$

# Computational Problem

- In a computational problem, the answers can be arbitrary strings. For example, factoring where the instances are (string representations of) positive integers and the solutions are (string representations of) collections of primes.
- A computational problem is represented as a **relation** consisting of all the instance-solution pairs, called a search relation. For example, factoring can be represented as the relation

$$R = \{(4, 2), (6, 2), (6, 3), (8, 2), (9, 3), (10, 2), (10, 5)\dots\}$$

which consist of all pairs of **(n, p)**, where p is a nontrivial prime factor of n.



# Computers



What is the **purpose** of inventing computers?

# Computers



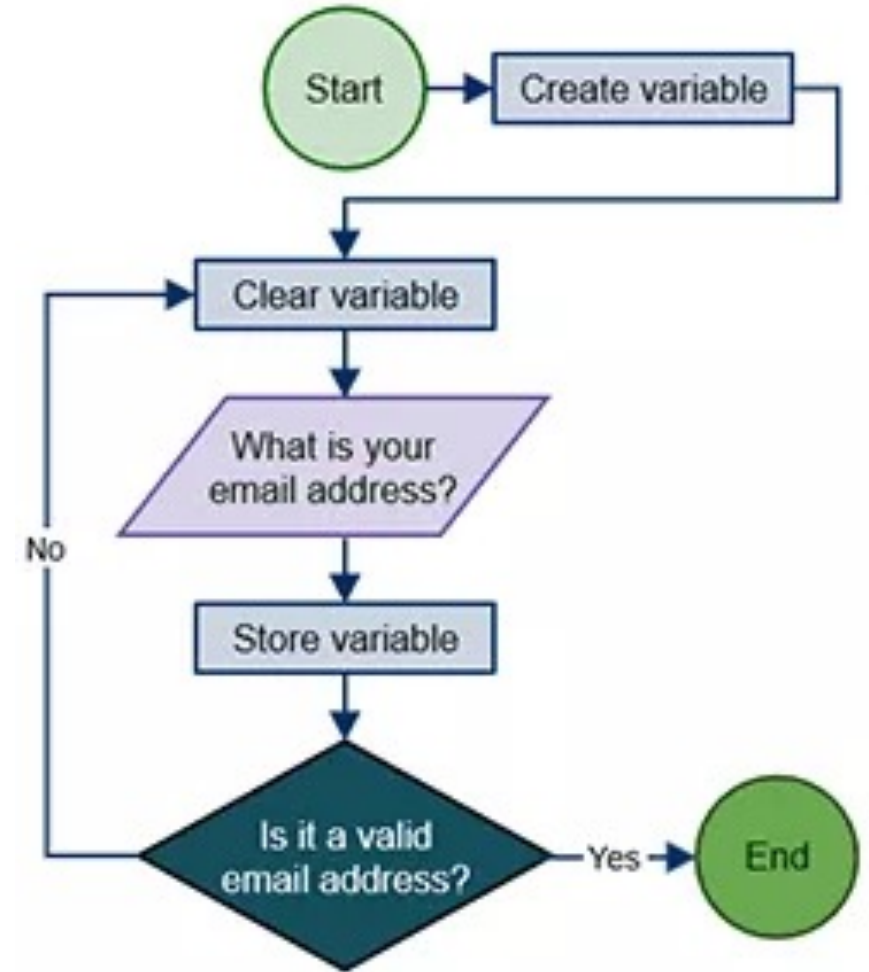
Can a brand-new computer solve our **specific** problems?

# Algorithms

An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically

- to **solve a class of specific problems**, or
- to perform a computation.

We run the algorithm in a computer which will find a solution for a given instance.



# Example: Sorting Algorithm

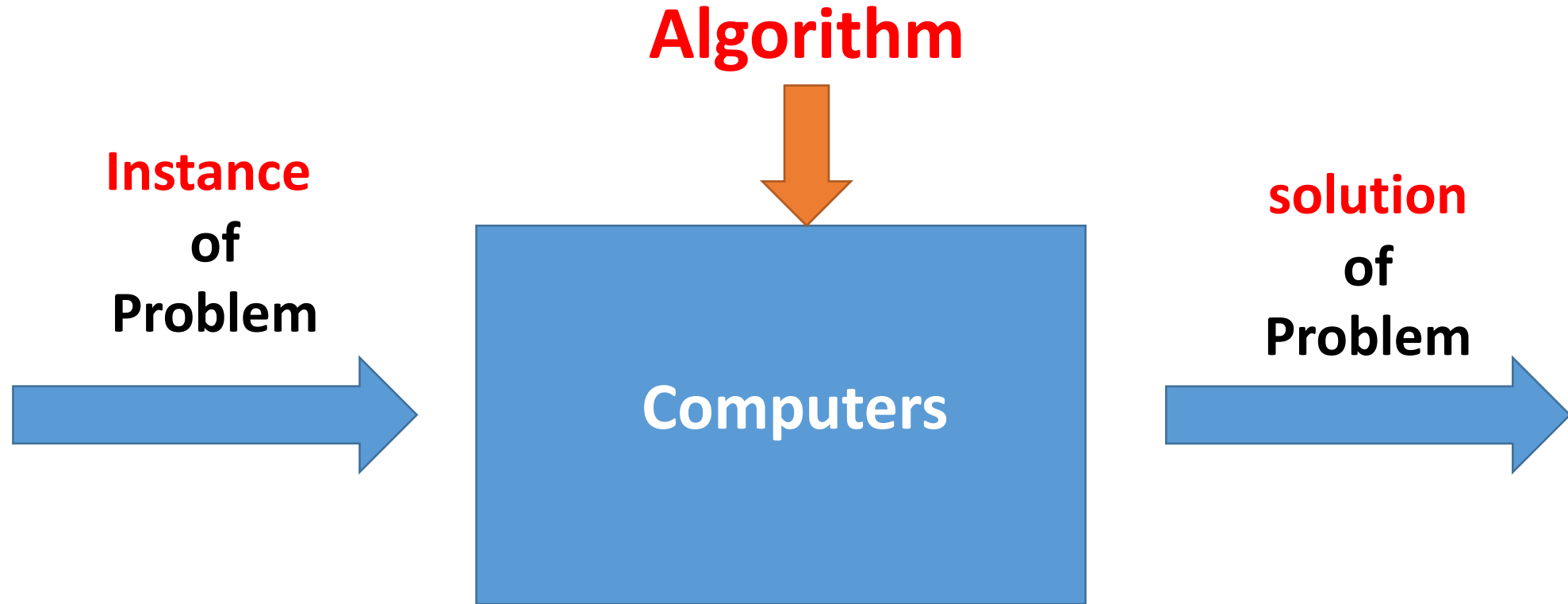
A sorting algorithm:

- Input elements of a list in a certain order
- Output a list in nondecreasing order

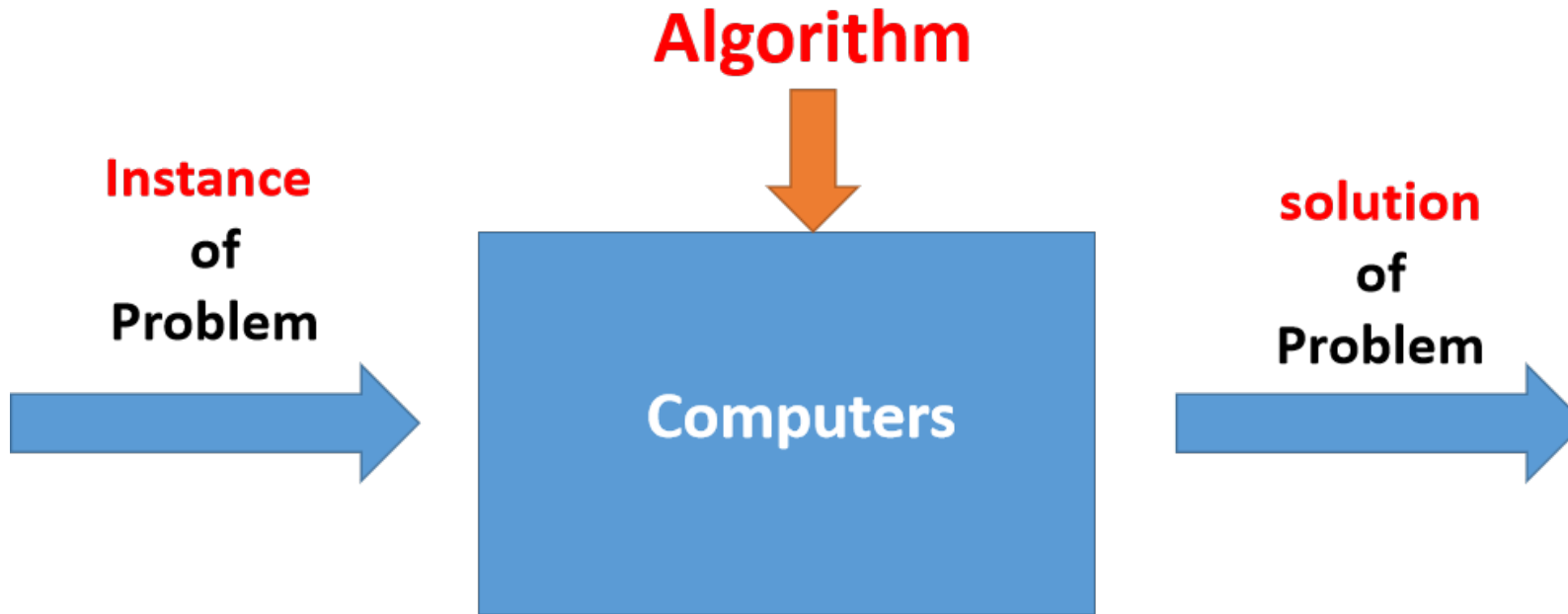
Input:  $L=(5,4,2,6,3)$

Output:  $L=(2,3,4,5,6)$

# Concepts in Complexity



# Algorithms: What do we care



1. Time cost
2. Space cost

# Comparison of Cost

A sorting algorithm:

- Input elements of a list in a certain order
- Output a list in nondecreasing order

Input:  $L=(5,4,2,6,3)$

Output:  $L=(2,3,4,5,6)$



The cost of time and space  
could increase

Input:  $L=(5,4,81,12,2,6,76,3,11)$

Output:  $L=(2,3,4,5,6,11,12,76,81)$

# Another Problem: Counting Order

Given instance  $I=(a_1, a_2, a_3, \dots, a_n)$  where  $a_i$  are integers

Output:  $k$  satisfying

$$a_1 \leq a_2 \leq \dots \leq a_k \text{ and } a_k > a_{k+1}$$

For example:

$$I_1=(1,3,4,2,4,5) \quad k=?$$

$$I_2=(2,3,4,5,7,9) \quad k=?$$

$$I_3=(9,1,3,5,6,7) \quad k=?$$



# Another Problem: Counting Order

Given instance  $I=(a_1, a_2, a_3, \dots, a_n)$  where  $a_i$  are integers

Output:  $k$  satisfying

$$a_1 \leq a_2 \leq \dots \leq a_k \text{ and } a_k > a_{k+1}$$

For example:

$$I_1=(1,3,4,2,4,5) \quad k=?$$

$$I_2=(2,3,4,5,7,9) \quad k=?$$

$$I_3=(9,1,3,5,6,7) \quad k=?$$

What can we observe about the cost?  $\text{length} + \text{case}$

# Best Case, Worst Case

In computer science, the **worst-case complexity** measures the resources (e.g. running time, memory) that an algorithm requires. It gives an **upper bound** on the resources required by the algorithm.

In computer science, the **best-case complexity** measures the resources (e.g. running time, memory) that an algorithm requires. It gives an **lower bound** on the resources required by the algorithm.

**For those instances with the same length:**

Worst Case  $I_2 = (2, 3, 4, 5, 7, 9)$

Best Case  $I_3 = (9, 1, 3, 5, 6, 7)$

# Average Case

In computational complexity theory, the average-case complexity of an algorithm is the amount of some computational resource (typically time) used by the algorithm, averaged over **all possible inputs**.

Let  $C(I)$  be the cost of instance  $I$ .

$$\text{Average Case} = \sum \text{Pr}[I_i] * C(I_i)$$

# Computational Complexity and Cryptography

- In computational complexity, we care the **worst case**.
- In cryptography, we care the **average case**.

Why? (workshop)

# Under the worst case

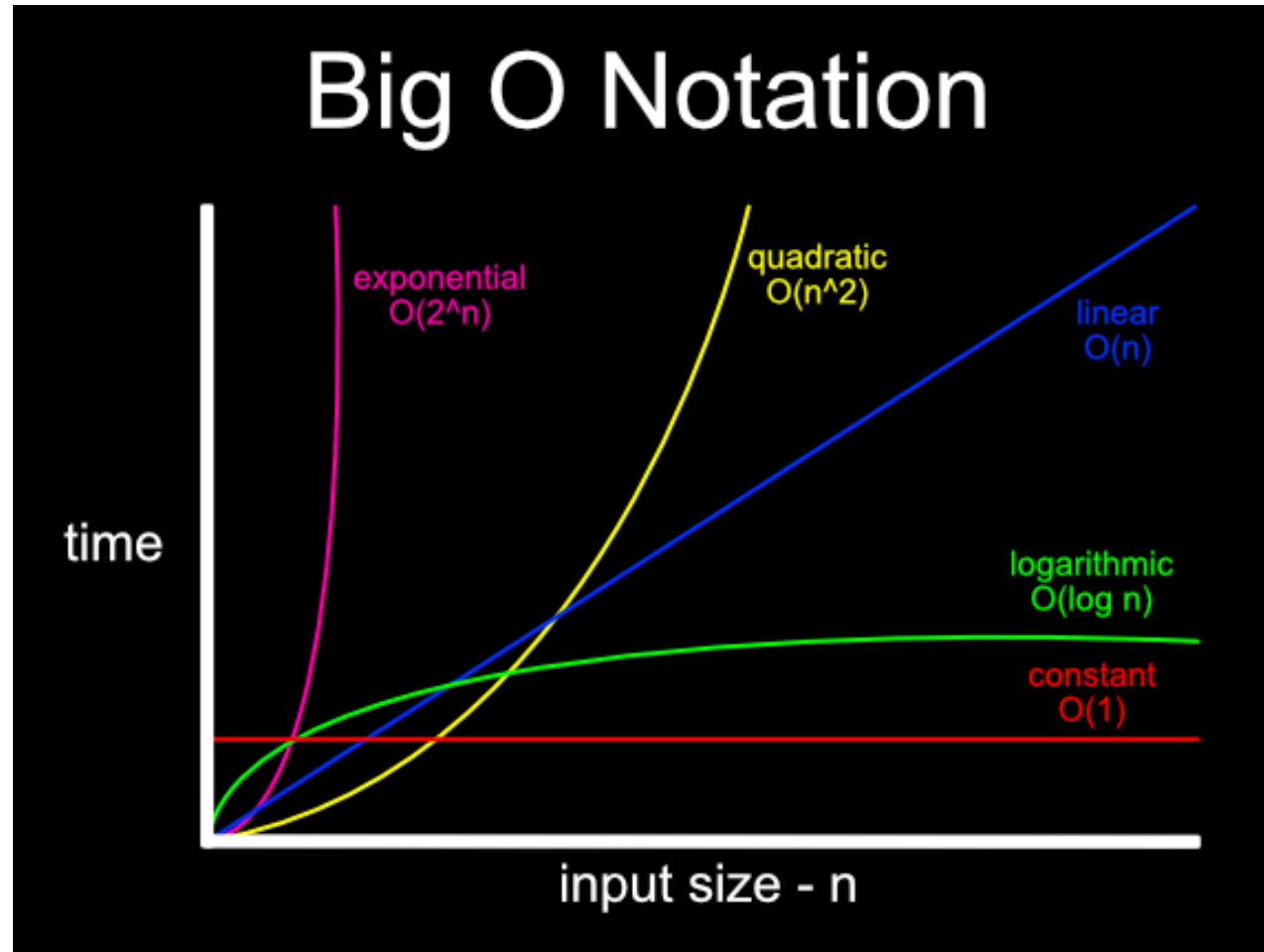
The cost can be denoted by

$C(1), C(2), C(3), C(4), \dots$

Here,  $C(n)$  denotes the cost of length- $n$  (worst-case) instance.

How fast will the cost increase?

# Big O notation: Up or Down



# Big O notation: Up or Down

Polynomial time: There are some values  $c$  and  $k$

$$f(n) \leq cn^k \text{ for any } n$$

An algorithm can solve problem A in polynomial time.

Exponential time: There are some values  $c$

$$f(n) \leq c2^n \text{ for any } n$$

An algorithm can solve problem A in exponential time.

# Big $\Omega$ notation: Up or Down

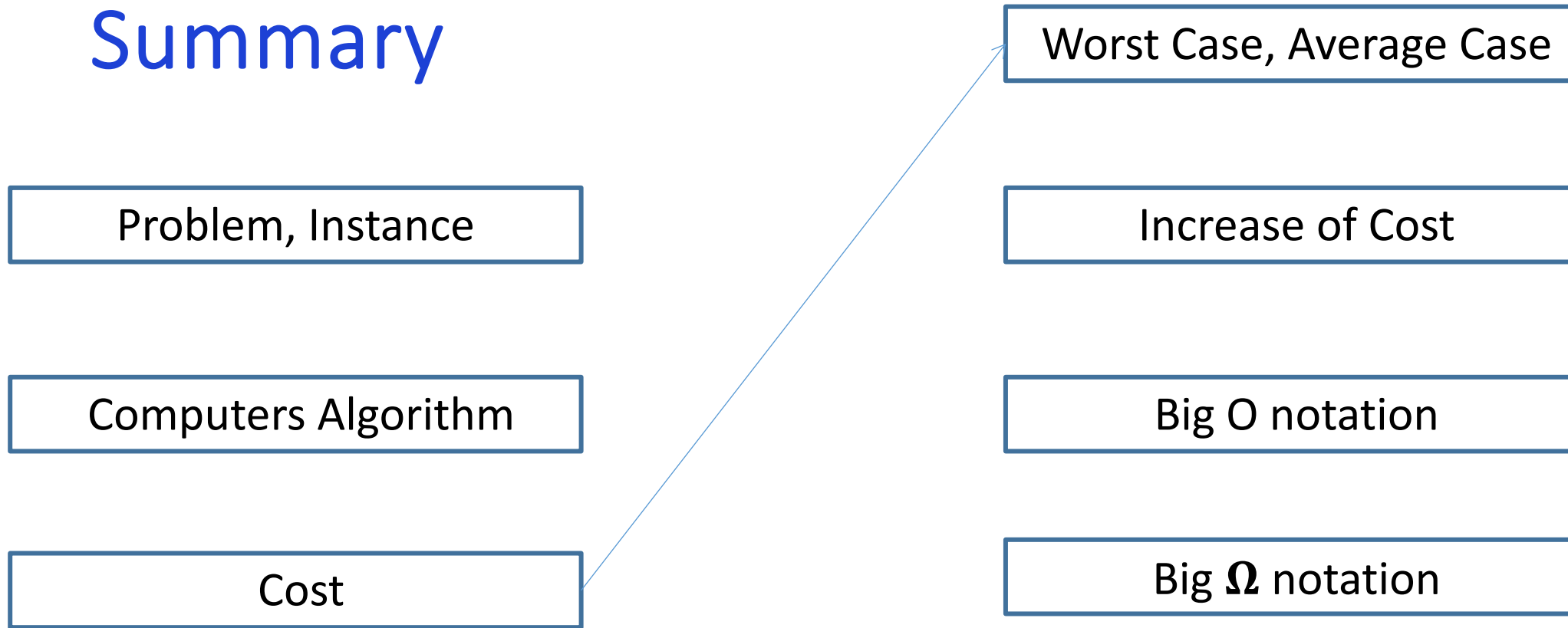
Exponential time: There are some values c

$$c2^n \leq f(n) \text{ for any } n$$

An algorithm solves problem A **with** exponential time.



# Summary



An algorithm can solve problem A **in** polynomial time.

An algorithm solves problem A **with** exponentitional time.

Question: How to answer this question?

Can algorithm A solve problem B in polynomial time?

Question: YES/NO

Can algorithm A solve problem B in polynomial time?

- Give the analysis
- Obtain the cost denoted by  $f(n)$
- Prove that  $f(n)$  is in polynomial time

- Give the analysis
- Obtain the cost denoted by  $f(n)$
- Show that  $f(n)$  is outside polynomial time

Question: How to answer this question?

Can problem B be solved in polynomial time?

Question: YES/NO

Can problem B be solved in polynomial time?

- Find an efficient algorithm
  - Obtain the cost denoted by  $f(n)$
  - Prove that  $f(n)$  is in polynomial time
- 
- Find the most efficient algorithm (suppose we can)
  - Obtain the cost denoted by  $f(n)$
  - Show that  $f(n)$  is outside polynomial time

# Comparison

- Can algorithm A solve problem B in polynomial time?

How efficient the algorithm is

- Can problem B be solved in polynomial time?

How hard the problem is

# Find an algorithm from (where)??

- An algorithm inside a computer that can correctly run the algorithm.
- What kinds of computer? Computer made by the human only or also those computers made by the alien?



# Turing Machine

A Turing machine is a **mathematical** model of computation that defines an **abstract** machine that manipulates symbols on a strip of tape according to **a table of rules**. (input-output based on table)





# Deterministic Turing Machine(wiki)

## Formal definition [\[ edit \]](#)

---

Following [Hopcroft & Ullman \(1979, p. 148\)](#), a (one-tape) Turing machine can be formally defined as a 7-tuple

$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$  where

- $\Gamma$  is a finite, non-empty set of *tape alphabet symbols*;
- $b \in \Gamma$  is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
- $\Sigma \subseteq \Gamma \setminus \{b\}$  is the set of *input symbols*, that is, the set of symbols allowed to appear in the initial tape contents;
- $Q$  is a finite, non-empty set of *states*;
- $q_0 \in Q$  is the *initial state*;
- $F \subseteq Q$  is the set of *final states* or *accepting states*. The initial tape contents is said to be *accepted* by  $M$  if it eventually halts in a state from  $F$ .
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  is a [partial function](#) called the *transition function*, where L is left shift, R is right shift. If  $\delta$  is not defined on the current state and the current tape symbol, then the machine halts;<sup>[\[22\]](#)</sup> intuitively, the transition function specifies the next state transited from the current state, which symbol to overwrite the current symbol pointed by the head, and the next head movement.

# Find an algorithm from (where)??

- An algorithm inside a computer that can correctly run the algorithm.

Computer made by the human



Computers made by the alien



Computers like Turing machine



# Non-Deterministic Turing Machine(wiki)

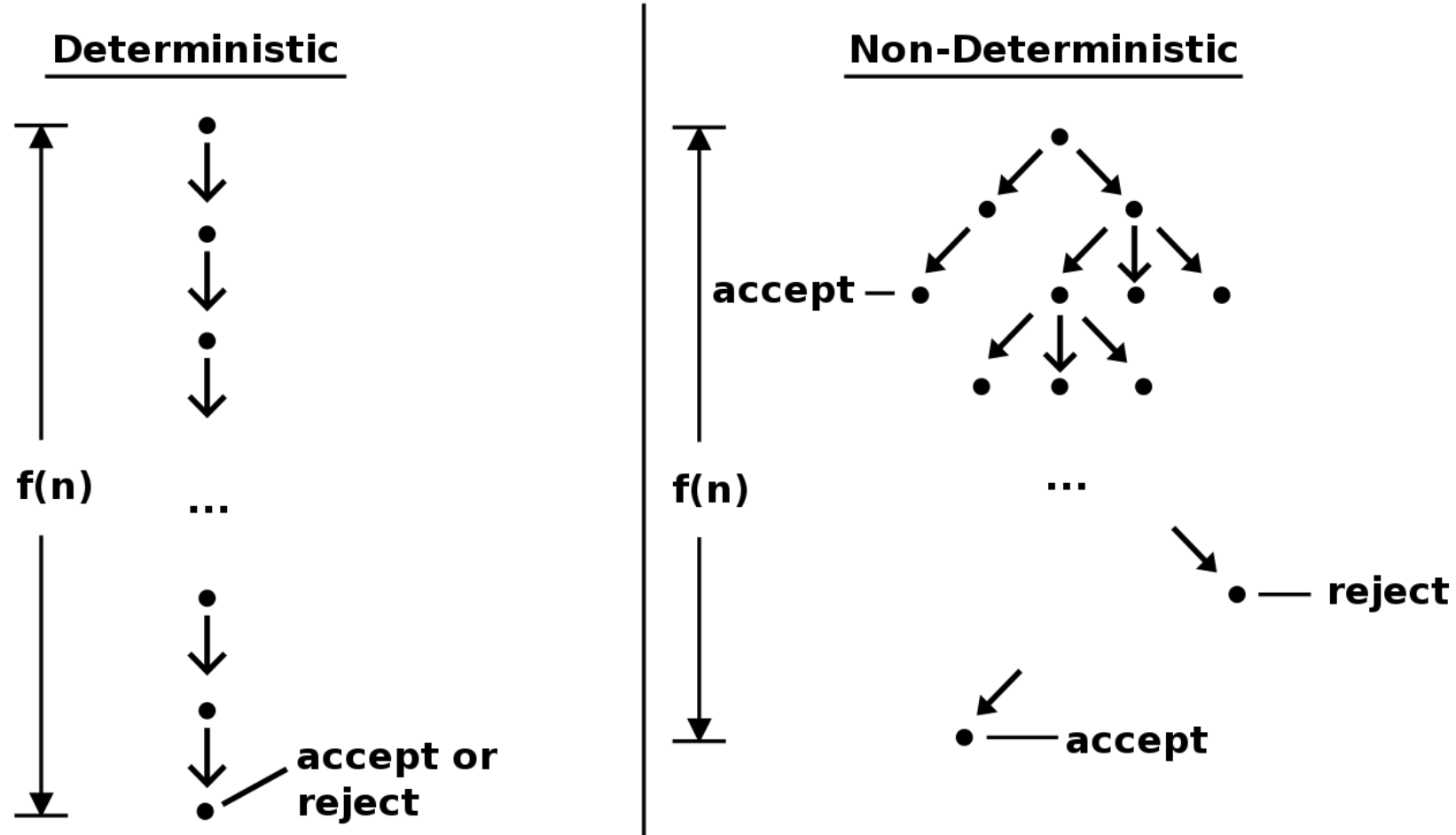
## Definition [\[ edit \]](#)

---

A nondeterministic Turing machine can be formally defined as a six-tuple  $M = (Q, \Sigma, \iota, \sqcup, A, \delta)$ , where

- $Q$  is a finite set of states
- $\Sigma$  is a finite set of symbols (the tape alphabet)
- $\iota \in Q$  is the initial state
- $\sqcup \in \Sigma$  is the blank symbol
- $A \subseteq Q$  is the set of accepting (final) states
- $\delta \subseteq (Q \setminus A \times \Sigma) \times (Q \times \Sigma \times \{L, S, R\})$  is a relation on states and symbols called the *transition relation*.  
 $L$  is the movement to the left,  $S$  is no movement, and  $R$  is the movement to the right.

# Deterministic vs. Non-Deterministic



# A Set of Decisional Problems: P and NP

- We define a set of decisional problems: **P**

Any decisional problem is inside the set P if it can be solved by a **deterministic** Turing machine in **P**olynomial time .

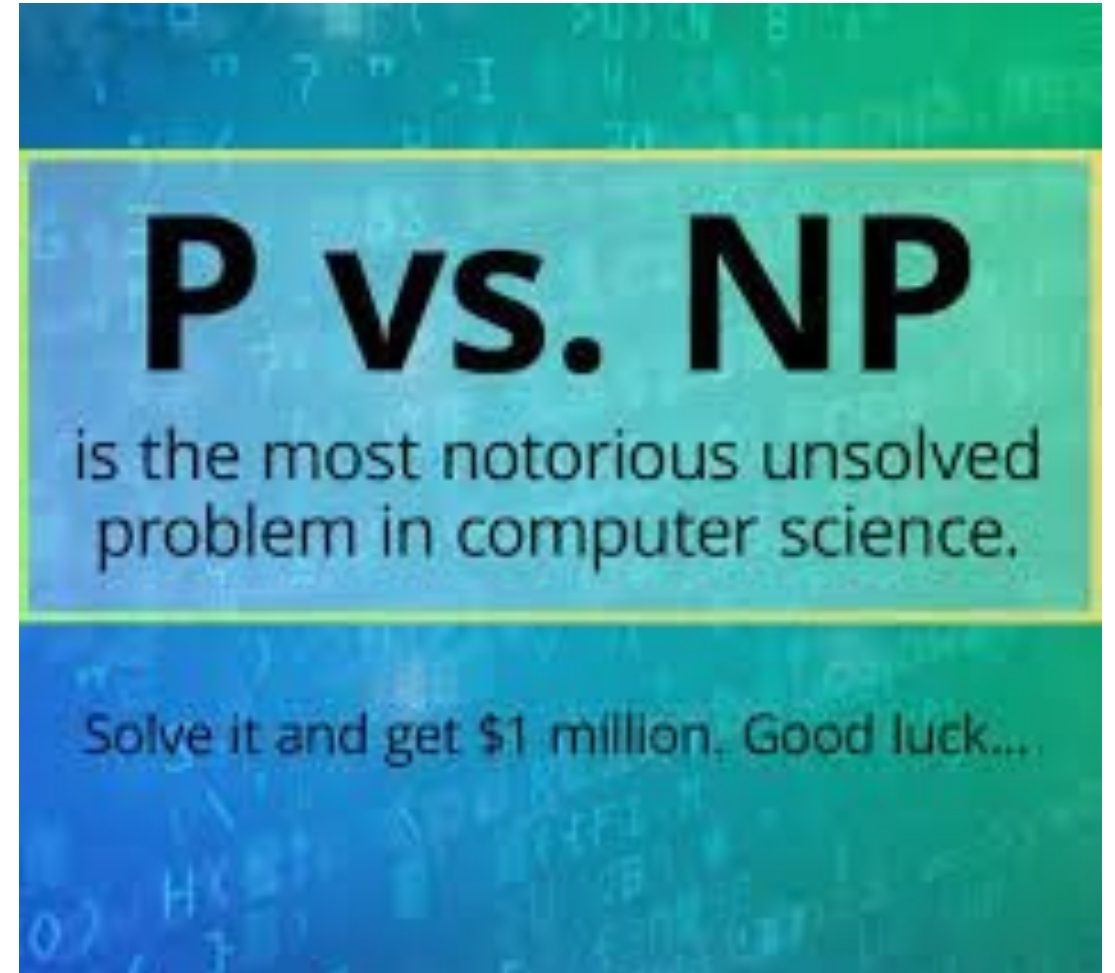
- We define a set of decisional problems: **NP**

Any decisional problem is inside the set NP if it can be solved by a **None-deterministic** Turing machine in **P**olynomial time.

NOTE: it is defined based on the worst case instance.

# A Set of Decisional Problems: P and NP

- If a problem is in P, it must be in NP.
- If a problem is in NP, **we still don't know it is in P or not.**

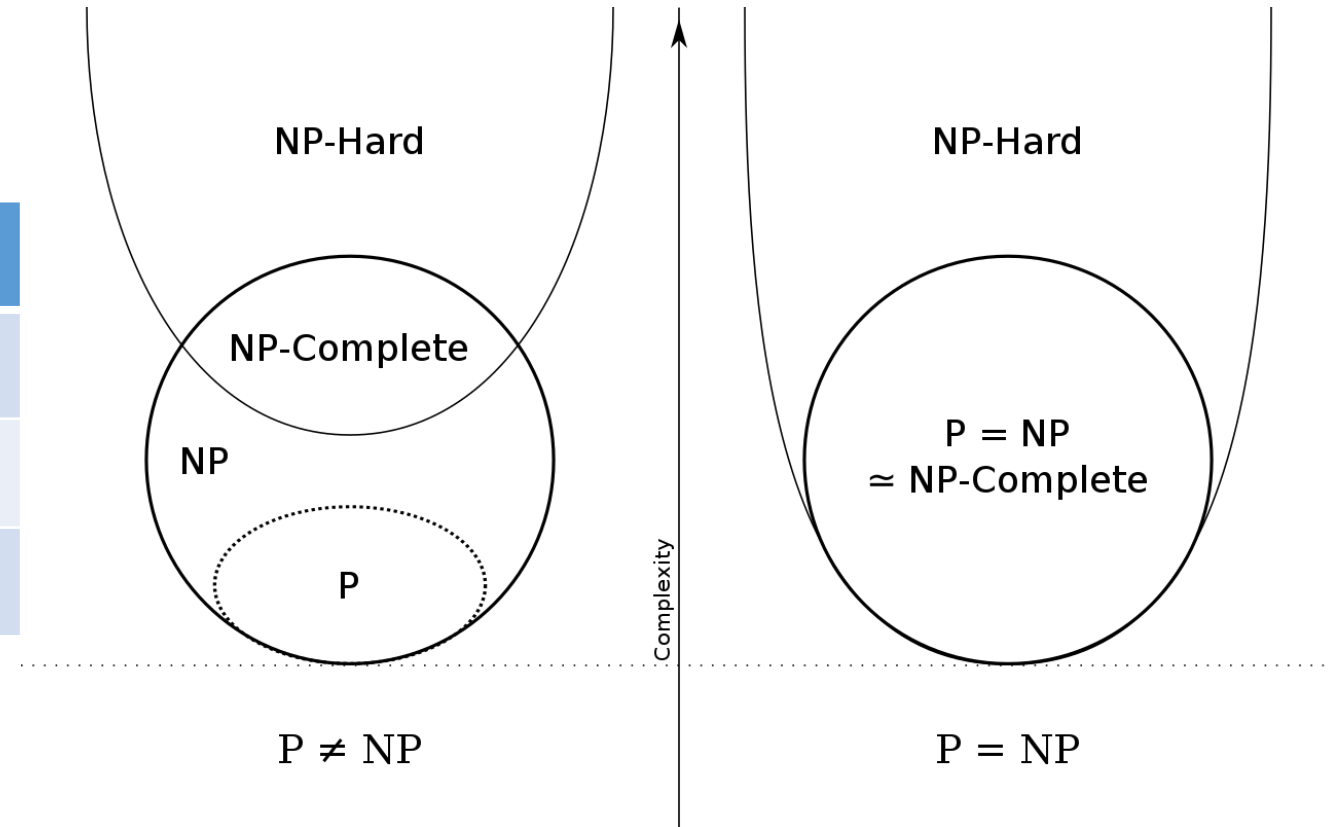


# Biref Summary

- Can algorithm A solve problem B in polynomial time?
- Can problem B be solved in polynomial time?
- Can problem B be solved by a DTM in polynomial time?
- Can problem B be solved by a NDTM in polynomial time?

# NP-Complete

|         | NP              |
|---------|-----------------|
| Hardest | NP-Complete     |
| Harder  | NP-Intermediate |
| Easy    | P               |





# NP-Complete: Definition

## Formal definition [\[edit\]](#)

---

See also: *formal definition for NP-completeness (article P = NP)*

A decision problem  $\mathcal{C}$  is NP-complete if:

1.  $\mathcal{C}$  is in NP, and
2. Every problem in NP is [reducible](#) to  $\mathcal{C}$  in polynomial time.<sup>[3]</sup>

$\mathcal{C}$  can be shown to be in NP by

Note that a problem satisfying c

A consequence of this definition  
in NP in polynomial time.

## Background [\[edit\]](#)

In computability theory and computational complexity theory, a **many-one reduction** is a reduction which converts instances of one decision problem  $L_1$  into instances of a second decision problem  $L_2$  where the instance reduced to is in the language  $L_2$  if the initial instance was in its language  $L_1$ .

verified in polynomial time.

satisfies condition 1.<sup>[4]</sup>

JTM, or any other [Turing-equivalent abstract machine](#)

---

# Reduction

Problem A is **reducible to** Problem B in polynomial time.

- Solving problem B can be transformed to solving problem A.
- Problem B is harder than problem A

# Reduction

Problem A is **reducible to** Problem B in polynomial time.

## Proof:

Suppose there is an algorithm C that can solve problem B.

- We are given a problem instance **I\_A** of problem A.
- We create a problem instance I\_B of problem B with the instance **I\_A**
- We can use the solution to I\_B to compute a solution to **I\_A**.

Done!

# Example-1

Suppose  $f(x)$  is a one-way permutation that is hard to compute  $x$  from  $f(x)$

Problem A: Given  $f(a)$ , compute  $f(a+10)$

Problem B: Given  $f(b)$ , compute  $f(b+1)$ .

Then, problem A is reducible to problem B. (How?)

## Example-2

Suppose  $f(x)$  is a one-way permutation that is hard to compute  $x$  from  $f(x)$

Problem A: Given  $f(a)$  and  $k$ , compute  $f(a+k)$

Problem B: Given  $f(b)$ , compute  $b$ .

Then, problem A is reducible to problem B. (How?)

# Example-3

Suppose  $f(x)$  is a one-way permutation that is hard to compute  $x$  from  $f(x)$

Problem A: Given  $f(a)$  , compute  $a$

Problem B: Given  $f(b)$  and  $f(c)$ , compute  $b+c$ .

Then, problem A is reducible to problem B. (How?)

## Example-3

Suppose  $f(x)$  is a one-way permutation that is hard to compute  $x$  from  $f(x)$

Problem A: Given  $f(a)$ , compute  $a$

Problem B: Given  $f(b)$  and  $f(c)$ , compute  $b+c$ .

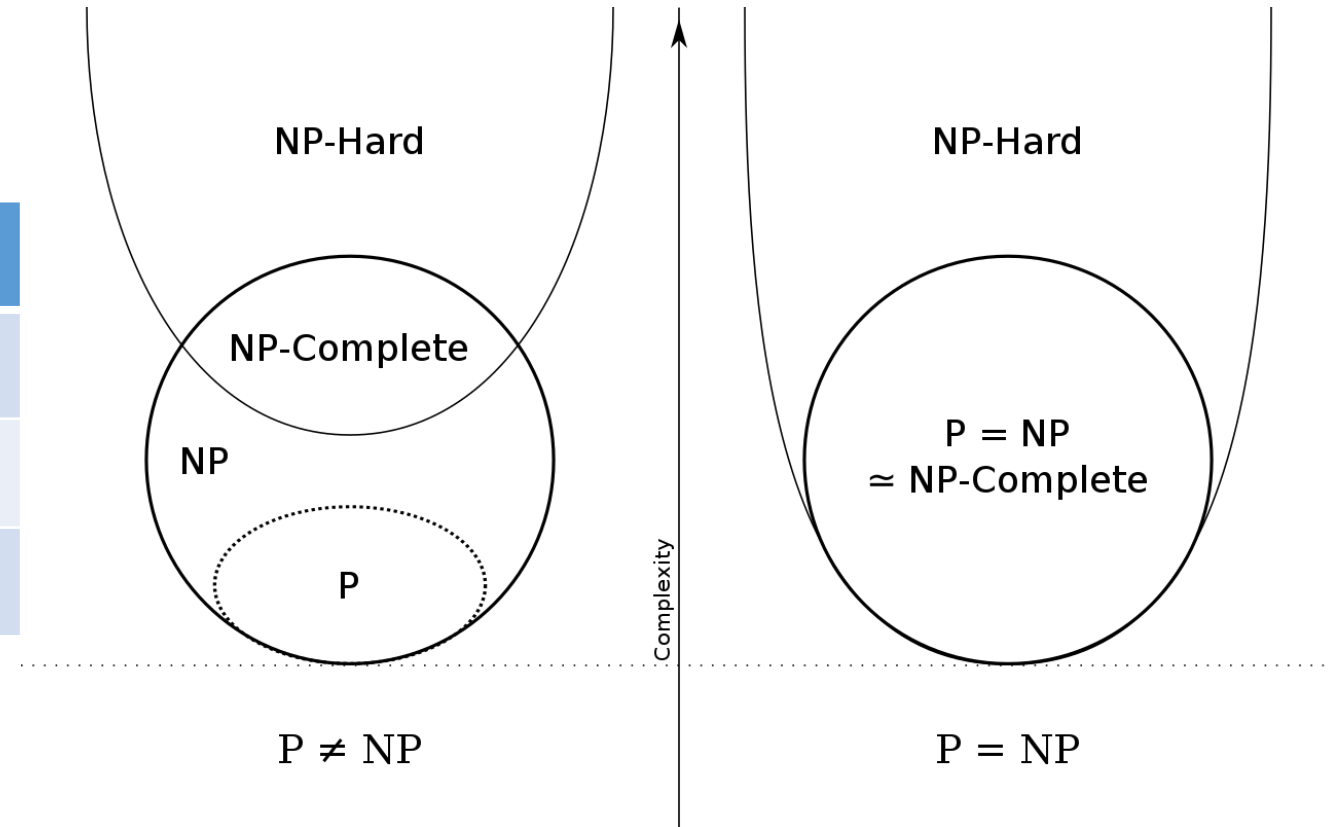
Then, problem A is reducible to problem B. (How?)

**Proof.** Suppose there is an algorithm  $C()$  that can solve problem B. Given as input  $f(a)$ , our task is to compute  $a$  with the help of algorithm  $C()$ .

1. We compute  $f(0)$ .
2. We run Algorithm  $C()$  with  $f(a)$  and  $f(0)$  as the input. The algorithm should return  $a+0=a$ .
3. We output  $a$  as the solution to  $f(a)$ , which solves problem A.

# NP-Complete

|         | NP              |
|---------|-----------------|
| Hardest | NP-Complete     |
| Harder  | NP-Intermediate |
| Easy    | P               |





# Summary of Concepts

- Problem, Computer, Algorithm
- Cost (worst-case, adverage-case, best-case)
- Big O and Big  $\Omega$
- Computational model (Turining Machine)
- Can problem B be solved by a DTM/NDTM in polynomial time
- P, NP, NP-Intermediate, NP-Complete
- Reduction and reducible

# From Computational Complexity to Cryptography

GAP



# From Computational Complexity to Cryptography

- Problem B cannot be solved by a DTM in polynomial time.



- Cryptosystem **cannot be broken** by an adversary in polynomial time.



- The cryptosystem is **computationally hard to be broken**.



- The cryptosystem is **secure**.

# Gap 1

- Problem B cannot be solved by a DTM in polynomial time.

In P and NP, the problem B is a **decisional** problem.

- Cryptosystem cannot be broken by an adversary in polynomial time.

In many cryptosystems, breaking the system = solve a **computational** problem, like breaking a ciphertext and obtaining its plaintext.

## Gap 2

- Problem B cannot be solved by a **DTM** in polynomial time.  
In P and NP, it is equivalent to a deterministic algorithm.
- Cryptosystem cannot be broken by **an adversary** in polynomial time.  
The adversary refers to a probabilistic algorithm.

# Deterministic vs Probabilistic

- In computer science, a **deterministic** algorithm is an algorithm that, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. Deterministic algorithms are by far the most studied and familiar kind of algorithm, as well as one of the most practical, since they can be run on real machines efficiently.
- A **randomized** algorithm is an algorithm that employs a degree of **randomness** as part of its logic or procedure. The algorithm typically uses uniformly random bits as an **auxiliary input** to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random determined by the random bits; thus either the running time, or the output (or both) are random variables.

# Deterministic vs Probabilistic

|        | Deterministic  | Probabilistic           |
|--------|----------------|-------------------------|
| Input  | instance       | instance & bit string   |
| Output | Always correct | correct with $\geq 2/3$ |

# Gap 3

- Problem B cannot be **solved** by a DTM in polynomial time.

It is referred to the worst case.

- Cryptosystem cannot be **broken** by an adversary in polynomial time.

It is referred to the average case.



# Workshop

# Question-1

In computational complexity, we care the **worst case**.  
In cryptography, we care the **average case**.

Why?

## Question-1

In public-key cryptography, there is a key generation algorithm. Each user can run the algorithm to generate a key pair  $(pk, sk)$ .

$pk$  is called public key while  $sk$  is called secret key.

Everyone can know  $pk$  but  $sk$  is kept secret by the user.

It should be hard to compute  $sk$  from  $pk$ .

# Question-1

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose hard means that computing will take 200 years.

It is **worst**-case hard to compute a secret key from a public key.

It is **best**-case hard to compute a secret key from a public key.

It is **average**-case hard to compute a secret key from a public key.

## Answer-1

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose hard means that computing will take 200 years.

It is **worst**-case hard to compute a secret key from a public key.

*There is one key pair and it will take 200 years for the adversary to compute its secret. The other key pairs might be very easy.*

## Answer-1

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose hard means that computing will take 200 years.

It is **best**-case hard to compute a secret key from a public key.

*All key pairs will take 200 years for the adversary to compute its secret. The min time cost is 200 years.*

## Answer-1

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose hard means that computing will take 200 years.

It is **average**-case hard to compute a secret key from a public key.

*The average cost will take 200 years for the adversary to compute its secret. Some of key pairs will take more than 200 while some of key pairs will take less than 200 years.*

## Question-2

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose easy means that computing will take 1 day.

It is worst-case **easy** to compute a secret key from a public key.

It is best-case **easy** to compute a secret key from a public key.

It is average-case **easy** to compute a secret key from a public key.



## Answer-2

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose easy means that computing will take 1 day.

It is worst-case **easy** to compute a secret key from a public key.

*There is one key pair and it will take 1 day for the adversary to compute its secret. It is a max time. The other key pairs might be very easy in seconds.*

## Answer-2

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose easy means that computing will take 1 day.

It is best-case **easy** to compute a secret key from a public key.

*All key pairs will take 1 day for the adversary to compute its secret.  
The min time cost is 1 day.*

## Answer-2

It should be hard for an adversary to compute  $sk$  from  $pk$ .

How to understand the following statements?

suppose easy means that computing will take 1 day.

It is average-case **easy** to compute a secret key from a public key.

*The average cost will take 1 day for the adversary to compute its secret. Some of key pairs will take more than 1 day such as 200 years while some of key pairs will take less than 1 hour.*