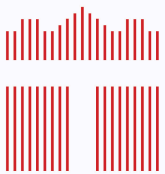
Slido: #ADA2020

CSIE 2136 Algorithm Design and Analysis, Fall 2020



National Taiwan University
國立臺灣大學

# NP Completeness - I

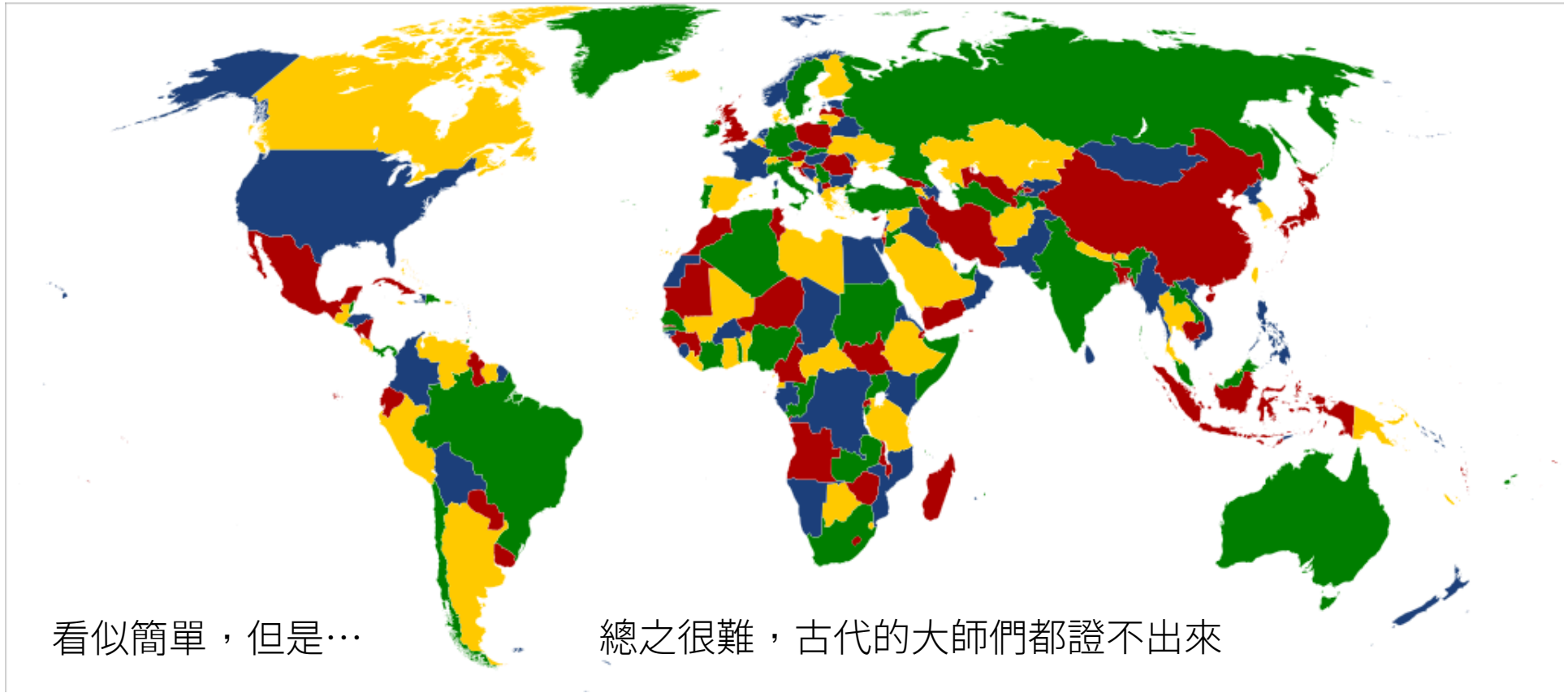
Hsu-Chun Hsiao

# 3-week Agenda

- NP-Completeness Overview
  - Warm up: four color problem
  - Decision vs. optimization
  - Complexity classes
  - Reduction
  - P-time solving vs. verification
- Proving NP-Completeness
  - Formula satisfiability problem
  - 3-CNF-SAT
  - The clique problem
  - The vertex-cover problem
  - The independent-set problem
  - Traveling salesman problem
  - Hamiltonian cycle
- Approximation algorithms
  - Vertex Cover
  - TSP
  - 3-CNF-SAT
- Randomized algorithms
  - Karger's min-cut algorithm
  - Probabilistic data structures

# 史上難題：四色問題 (Four Color Problem)

- 只能使用四種顏色，使每兩個鄰接區域的顏色都不一樣



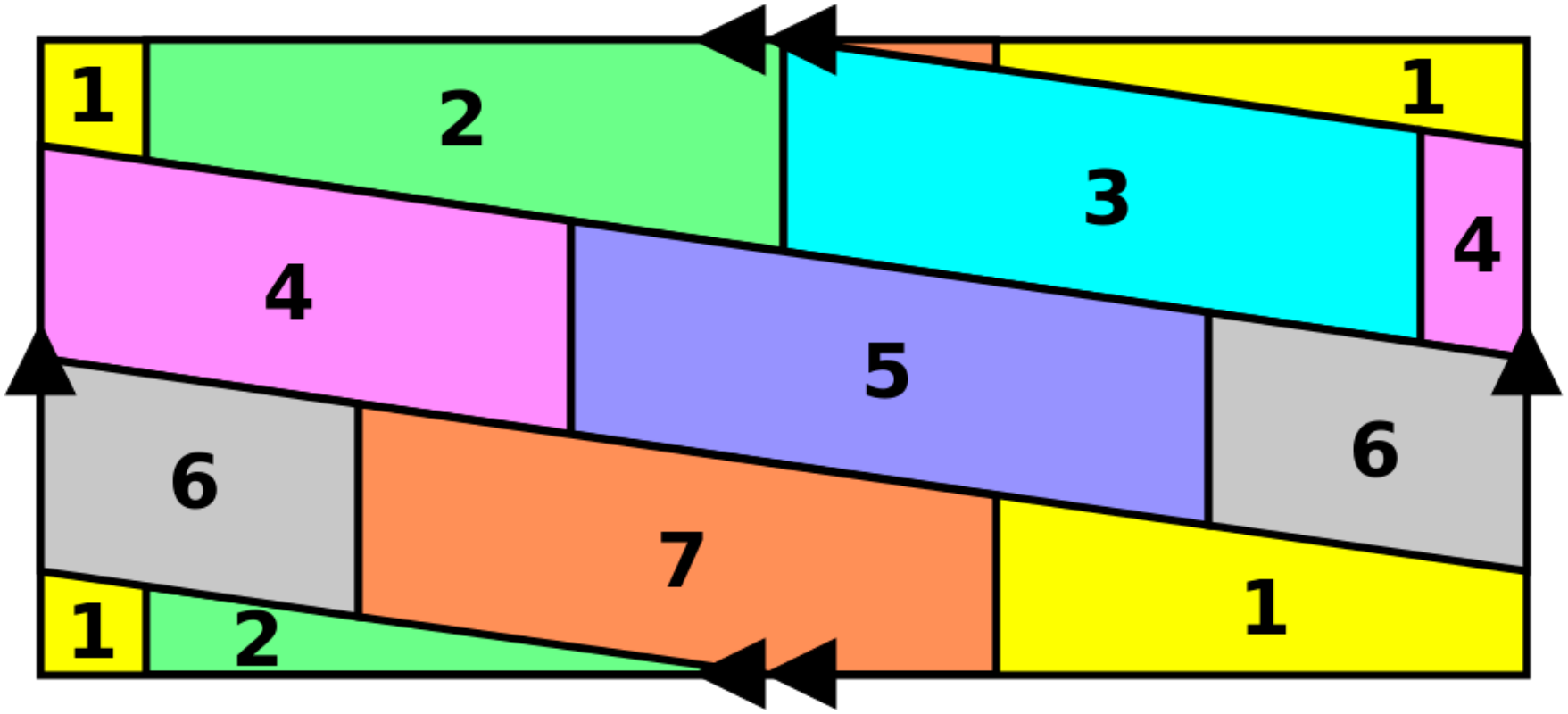
看似簡單，但是…

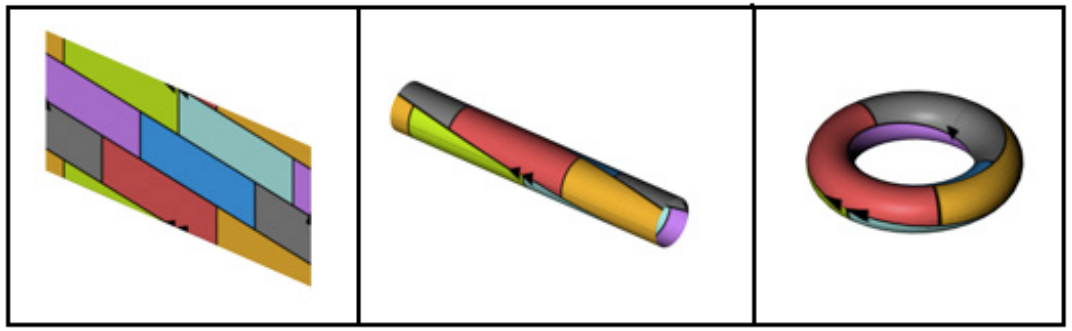
總之很難，古代的大師們都證不出來

# 一百多年後的四色定理

- Finally proven (with the help of computers) by Kenneth Appel and Wolfgang Haken in 1976
  - Their algorithm runs in $O(n^2)$ time
- First major theorem proved by a computer
- Open problems remain…
  - Linear time algorithms to find a solution
  - Concise, human-checkable, mathematical proofs

# 沒那麼簡單之一



# 沒那麼簡單之二


Bundesrepublik Deutschland
Büsingen
Gottmadingen
Kanton Schaffhausen
Stadt Schaffhausen
Gailingen
Diessenhofen
Kanton Zürich
Stein am Rhein
Kanton Thurgau
Schweiz

## 平面圖k色問題 (Planar k-colorability)

Given a planar graph G (e.g., a map), can we color the vertices with $k$ colors such that no adjacent vertices have the same color?

Q: Can you color this map using only 2 colors? Using only 3 colors?

2 colors: No

3 colors: Yes

## 平面圖k色問題 (Planar k-colorability)

Given a planar graph G (e.g., a map), can we color the vertices with $k$ colors such that no adjacent vertices have the same color?

Q: Can we efficiently solve the planar k-colorability problem when $k = 1$? $k = 2$? $k = 3$? $k \geq 4$?

$k = 1$: trivial

$k = 2$: trivial

$k = 3$: NP complete

$k = 4$: always possible because of the four-color theorem

# Decision Problems vs. Optimization Problems

## Optimization problems

Each feasible solution to the problem has an associated value, and we wish to find a feasible solution with the best value (i.e., maximum or minimum)

- Example
  - MST-OPT: Given a graph $G = (V, E)$, find the minimum spanning tree of $G$
  - KNAPSACK-OPT: Given a knapsack of capacity $C$ and a set of objects with weights and values, fill the knapsack so as to maximize the total value

## Decision problems

The answer to the problem is simply “yes” or “no” (or “1” or “0”)

- Examples
  - MST: Given a graph $G = (V, E)$ and a bound $K$, is there a spanning tree with a cost at most $K$?
  - KNAPSACK: Given a knapsack of capacity $C$, a set of objects with weights and values, and a target value $V$, is there a way to fill the knapsack with at least $V$ value?
- Technique to convert an optimization problem to a related decision problem: imposing a (lower or upper) bound on the value to be optimized

# Why focusing on decision problems?

- Every optimization problem has a decision version that is no harder than* the optimization problem.
- In fact, they have equal* computational difficulty.

* In the sense that they are in the same complexity class; we will explain more later

# Why focusing on decision problems?

- Every optimization problem has a decision version that is no harder than* the optimization problem.
- In fact, they have equal* computational difficulty.

$P_{opt}$: given a graph, find the length of the shortest path

$A_{opt}$ solves $P_{opt}$

$P_{dec}$: given a graph, determine whether there is a path $\leq k$

$A_{dec}$ solves $P_{dec}$

Q: Can we use $A_{opt}$ to solve $P_{dec}$? Use $A_{dec}$ to solve $P_{opt}$?

Using $A_{opt}$ to solve $P_{dec}$: check if the optimal value $\leq k$, constant overhead

Using $A_{dec}$ to solve $P_{opt}$: apply binary search on the value range, logarithmic overhead

# Complexity Classes

# Algorithm design

- We have learned several algorithmic design methods to solve computational problems efficiently.
  - Divide and conquer, dynamic programming, greedy…

- However, we have also encountered some hard problems without known efficient algorithms
  - Efficient = tractable = polynomial running time (in the size of input)
  - K-colorability, Hamiltonian, knapsack, …

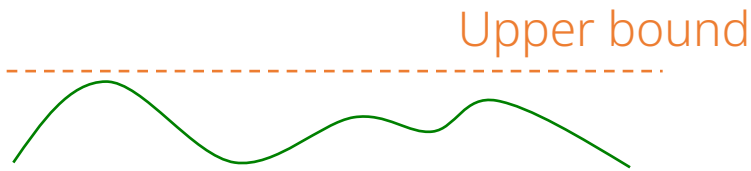
# Know your enemy

- Can we decide whether a problem is “too hard to solve” before investing our time in solving it?
- Idea: Decide which complexity classes the problem belongs to via reduction
  - 已知問題 A 很難。若能證明問題 B 至少跟 A 一樣難，那麼問題 B 也很難。

# To solve, or not to solve?

## Algorithm design

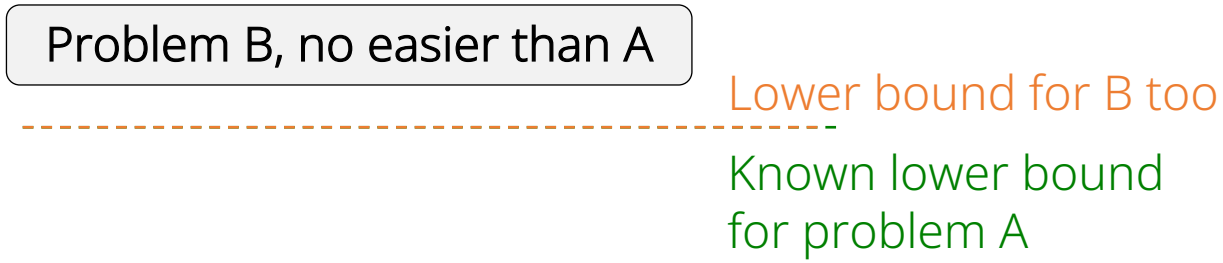
- Design algorithms to solve computational problems
- Mostly concerned with upper bounds on resources




E.g., Bellman-Ford is designed to find shortest paths in $O(VE)$ time

## Computational complexity theory

- Classify problems based on their difficulty and identify relationships between those classes
- Mostly concerned with lower bounds on resources




E.g., Solving Knapsack is no easier than solving SAT, which is known intractable, so Knapsack is intractable too

# Complexity classes

- A complexity class is defined as a set of problems of related resource-based complexity
  - Resource = time, memory, communication, …
- This lecture focuses on decision problems and time resource

# Complexity classes

- Class P: class of problems that can be solved in $O(n^k)$
  - $O(n^k)$ means polynomial in the input size ($k$ is a constant)
  - Problems in P are considered tractable
- Class NP: class of problems that can be verified in $O(n^k)$
  - NP = Nondeterministic Polynomial

Q: Is P $\subseteq$ NP? Is NP $\subseteq$ P?

P $\subseteq$ NP: Yes, because a problem solvable in polynomial time is verifiable in polynomial time as well

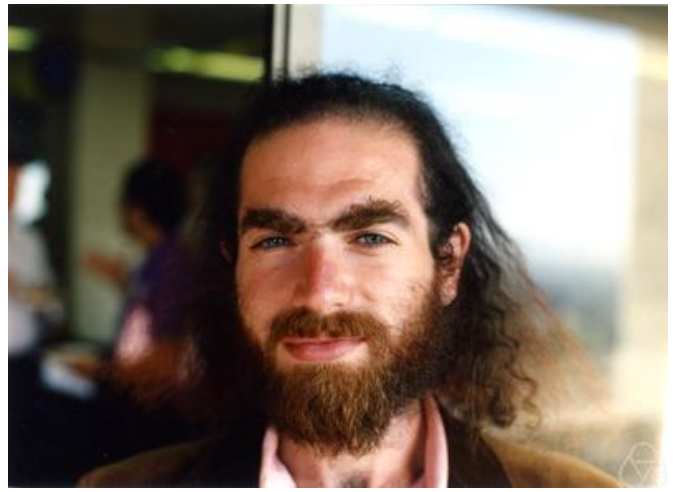
NP $\subseteq$ P: We don't know. If so, P = NP.

# Complexity classes

A question worth $1,000,000: Is P = NP?

# Millennium Prize

- Solve one and get $1,000,000!
  - P versus NP problem
  - Hodge conjecture
  - Riemann hypothesis
  - Yang-Mills existence and mass gap
  - Navier-Stokes existence and smoothness
  - Birch and Swinnerton-Dyer conjecture
  - Poincaré conjecture (solved by Grigori Perelman)



*Grigori Perelman*
Fields Medal (2006), declined
Millennium Prize (2010), declined

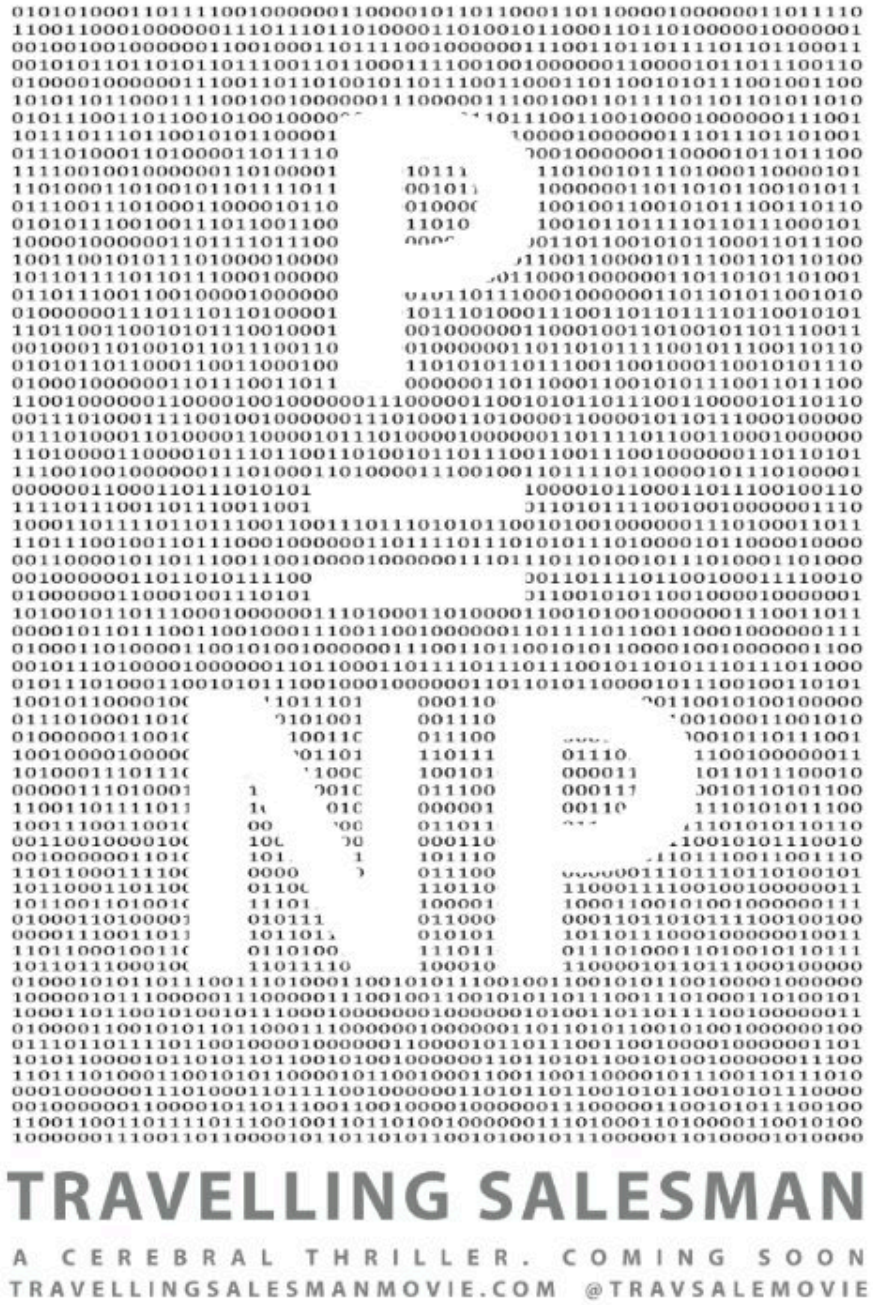
# Implications of P=NP

Problems that are verifiable => solvable

Public-key cryptography will be broken

"If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss…"
– Scott Aaronson, MIT

Widespread belief in P ≠ NP




# “Travelling Salesman” (2012)

A movie about P = NP

Best Feature Film in Silicon Valley Film Festival 2012

http://www.travellingsalesmanmovie.com/

# Complexity classes

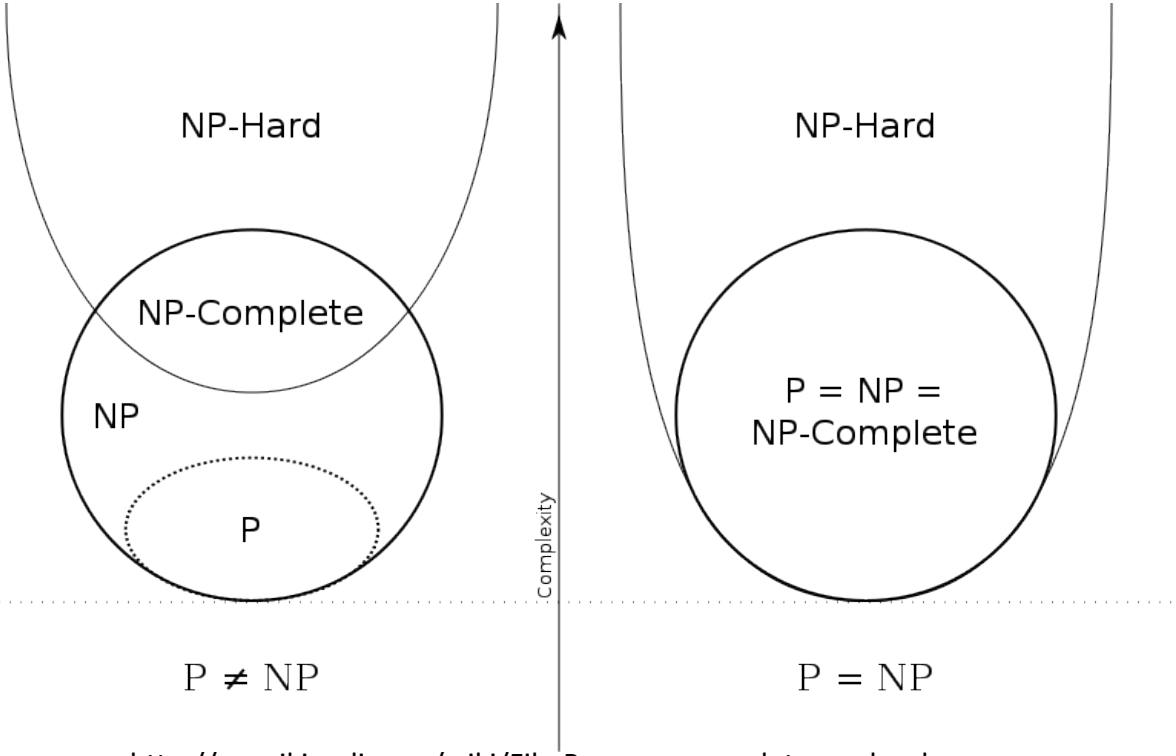
**Class P**: class of problems that can be solved in $O(n^k)$

**Class NP**: class of problems that can be verified in $O(n^k)$

**Class NP-hard:** a class of problems that are "at least as hard as the hardest problems" in NP

- Hardness relationship can be determined via polynomial-time reduction

**Class NP-complete (NPC)**: class of problems in both NP and NP-hard




http://en.wikipedia.org/wiki/File:P_np_np-complete_np-hard.svg

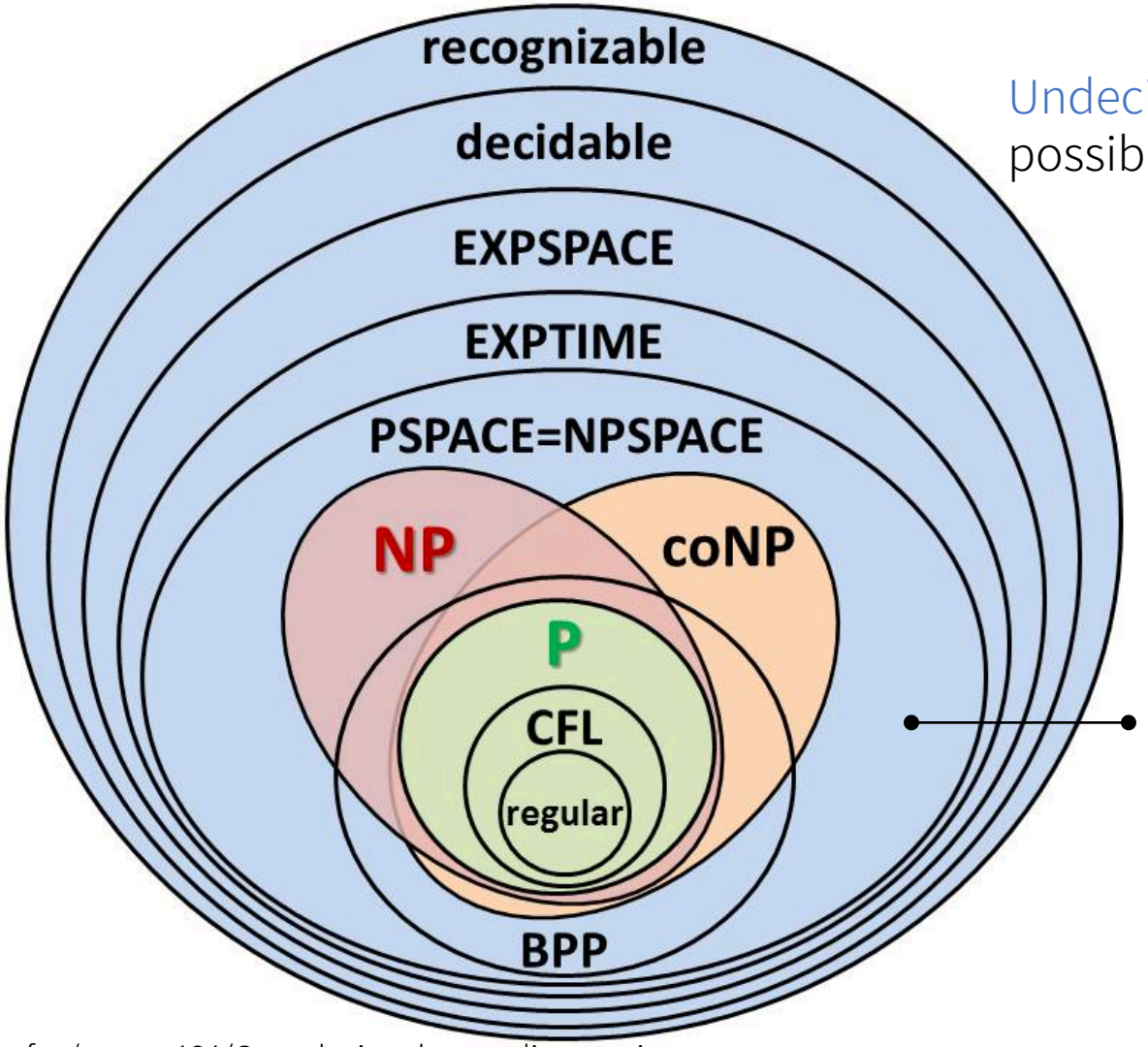
# The importance of the NP-Complete class

- Q: Why do we focus on studying NP-complete problems?
- A: Because they're the hardest in NP and solving one NPC problem can lead to solving all NP problems.
  - 是 NP 問題裡面最難的，解決一個就解決全部的 NP 問題
  - "Complete" in the sense that solving any problem in NPC in polynomial time => solving every problem in NP in polynomial time

# The importance of the NP-Complete class

- Q: Why do we focus on studying NP-complete problems?
- A: It's wildly believed that NPC problems have no polynomial-time solution, so they can be good reference points to judge whether an unknown problem is solvable in polynomial time.
  - 普遍相信 NPC 問題不存在 polynomial-time 解，因此可以當作參考物判斷一個未知問題是否無 polynomial-time 解
  - We can decide whether a problem is "too hard to solve" by showing it is no easier than a NPC problem
  - Then we can turn our focus to designing approximate algorithms or solving special cases

# More complexity classes



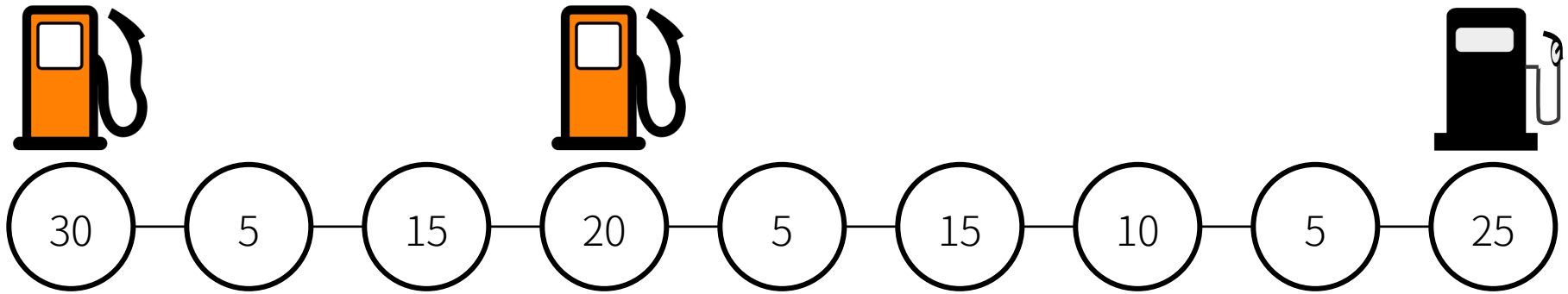

Undecidability. No algorithm possible, such as Halting problem.

Example: chess and Go

http://www.cse.psu.edu/~sofya/cmpsc464/Complexity-classes-diagram.jpg

# A PSPACE-complete problem: Competitive facility location

- **Problem:** Two gas station companies compete for the market in several zones, and according to the law, two stations CANNOT be in the same or adjacent zone. They take turns to open one station at a time. Is there a strategy for the 2nd player to select nodes with a total value of at least $B$ regardless of what the 1st player do?
- The problem is PSPACE-complete: no known efficient algorithm to even verify a given answer!



## Halting Problem

Given a program and an input, determine whether the program will finish running, or continue to run forever.

到底會一直跑一直跑一直跑一直跑
還是會停下來，要怎麼知道呢？

# Theorem: Halting problem is undecidable

## Proof

- Let's show that Halting problem is undecidable by constructing a counterexample
  - Suppose $h$ can determine whether a program $p$ halts on input $x$
    - $h(p, x)$ = return ($p$ halts on input $x$)
  - Define $g(p)$ = if $h(p, p)$ is $0$ then return $0$ else $HANG$
  - => $g(g)$ = if $h(g, g)$ is $0$ then return $0$ else $HANG$
- Consider whether $g$ halts on $g$, both cases contradict the assumption:
  - **$g$ halts on $g$**: then $h(g, g) = 1$, which would make $g(g)$ hang
  - **$g$ does not halt on $g$**: then $h(g, g) = 0$, which would make $g(g)$ halt

# 差一點 差很多之complexity class

猜猜以下每組中哪一個in P？

| | | |
|---|---|---|
| Shortest simple path | vs. | Longest simple path |
| Euler tour | vs. | Hamiltonian cycle |
| LCS with two input sequences | vs. | LCS with arbitrary input sequences |
| Degree-constrained spanning tree | vs. | Minimum spanning tree |

# How about these games?

Sudoku

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | | | | | | 3 | 7 |
| 6 | 2 | | | | | | 9 | 8 |
| | | 5 | 9 | | 2 | 1 | | |
| | | 2 | 7 | | 3 | 4 | | |
| | | | | 5 | | | | |
| | | 1 | 4 | | 6 | 9 | | |
| | | 4 | 3 | | 8 | 6 | | |
| 2 | 5 | | | | | | 4 | 3 |
| 9 | 6 | | | | | | 1 | 2 |

Candy Crush

MineSweeper

# 別解問題の計算量と完全性、およびパズルへの応用

八登 崇之
東京大学大学院理学系研究科
情報科学専攻
yato@is.s.u-tokyo.ac.jp

瀬田 剛広
東京大学大学院情報理工学系研究科
コンピュータ科学専攻
seta@is.s.u-tokyo.ac.jp

**概　要**

問題 Π に対する別解問題 (ASP) というのは、Π のインスタンス $x$ とそれに対する 1 つの解 $s$ が与えられた時に、$x$ の $s$ 以外の解を求める問題のことである。(新しい問題クラスとしての ASP の概念は Ueda と Nagao による。) 本論文では $n$ 個の解が与えられた時にもう 1 つの解を求める問題 ($n$-ASP) について考察する。特に、対応する解への変換も多項式時間で行えるような多項式時間 parsimonious 還元に関する完全性である ASP 完全性について考察する。

応用として、本論文では 3 つの有名なパズル、スリザーリンクとカックロ (クロスサム) とナンバープレース (数独) についてその ASP 完全性 (NP 完全性を含意する) を示す。

# Complexity and Completeness of Finding Another Solution and Its Application to Puzzles

Takayuki YATO
Department of Information Science
Graduate School of Science
The University of Tokyo
yato@is.s.u-tokyo.ac.jp

Takahiro SETA
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo
seta@is.s.u-tokyo.ac.jp

**Abstract**

The Another Solution Problem (ASP) of a problem Π is the following problem: for a given instance $x$ of Π and a solution $s$ to it, find a solution to $x$ other than $s$. (The notion of ASP as a new class of problems was first introduced by Ueda and Nagao.) In this paper we consider $n$-ASP, the problem to find another solution when $n$ solutions are given. In particular we consider ASP-completeness, the completeness with respect to the parsimonious reductions which allow polynomial-time transformation of solutions.

As an application, we prove the ASP-completeness (which implies NP-completeness) of three popular puzzles: Slither Link, Cross Sum, and Number Place.

candycrush.isnphard.com

# Bejeweled, Candy Crush and other match-three games are (NP-)Hard!

---

## What is this all about?

This is an implementation of the reduction provided in the paper *Bejeweled, Candy Crush and other Match-Three Games are (NP)-Hard* which has been accepted for presentation at the *2014 IEEE Conference on Computational Intelligence and Games (CIG 2014)*. To find more about what NP-Hard means you can read this blog post or the corresponding page on Wikipedia.

## About the authors

We are an italian group of three people: Luciano Gualà, Stefano Leucci, and Emanuele Natale. The former is a researcher, while the others are Ph.D. students. We had the weird idea to spend our weekends proving that Candy Crush Saga is NP-Hard. We also thought that it was nice to put online an implementation of our hardness reduction... so here it is!

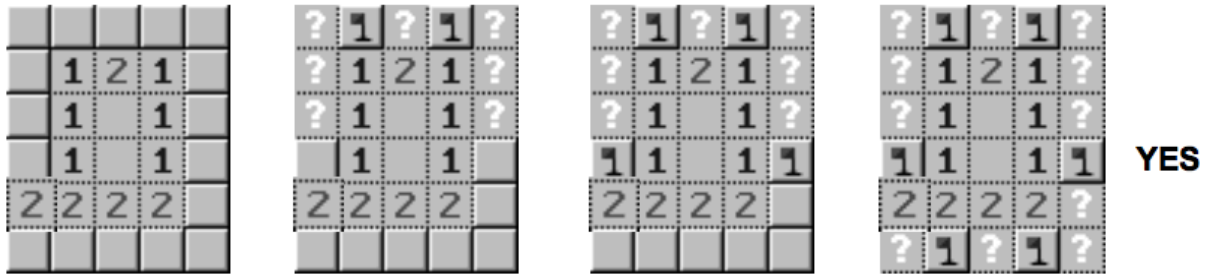
## Rules

Swap two adjacent gems in order to match three or more gems of the same kind. The matched gems will pop, and the gems above will fall. It is possibile to have chains of pops.

For a complete understanding of what's going on please read the paper on ArXiv.
**In a nutshell (for those "tl;dr" folks):** you can swap one or two gems on each choice wire from the top one to the bottom one, then you have to traverse the goal wire to reach the goal gem. Popping a wire means setting the corresponing variable to true.

# Minesweeper consistency problem is also NP-complete

- **Minesweeper consistency problem**: Given a state of what purports to be a Minesweeper game, is it logically consistent?

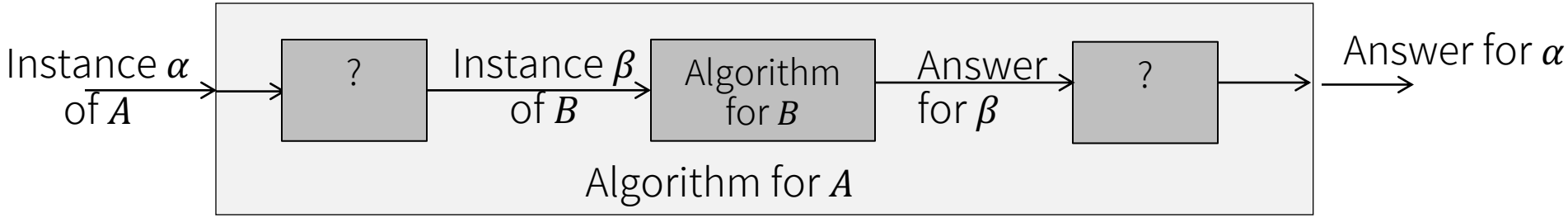



http://www.cs.princeton.edu/~wayne/cs423/lectures/np-complete-4up.pdf

# Reduction

# Reduction

- **General concept**: Problem $A$ reduces to problem $B$ if we can use an algorithm that solves $B$ to help solve $A$

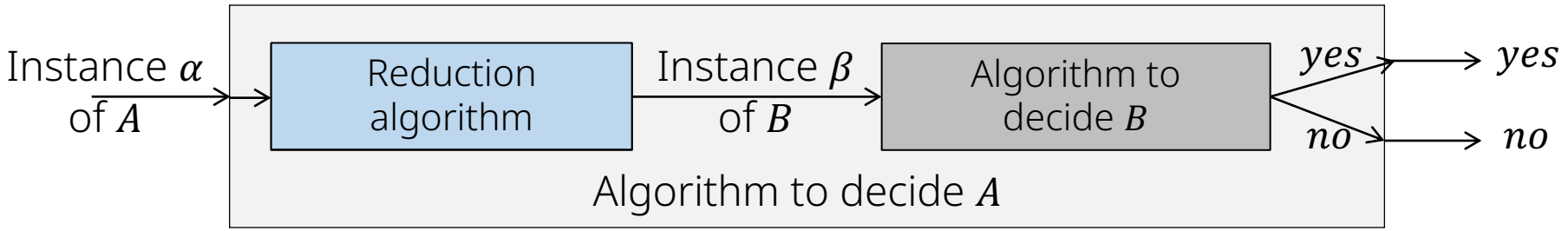



Complexity of the Algorithm for $A$?

# Reduction

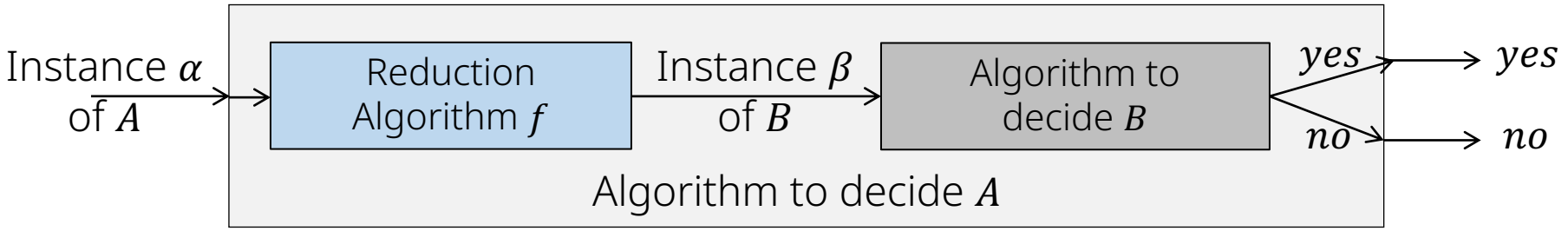
- A reduction is an algorithm for transforming one problem instance into another
- We focus on decision problems here
- For all $\alpha$, $AlgA(\alpha) = 1$ iff $AlgB(\beta) = 1$
- $A \leq_p B$ means…
  - $A$ can be reduced to $B$ in polynomial time
  - $A$ is no harder than $B$

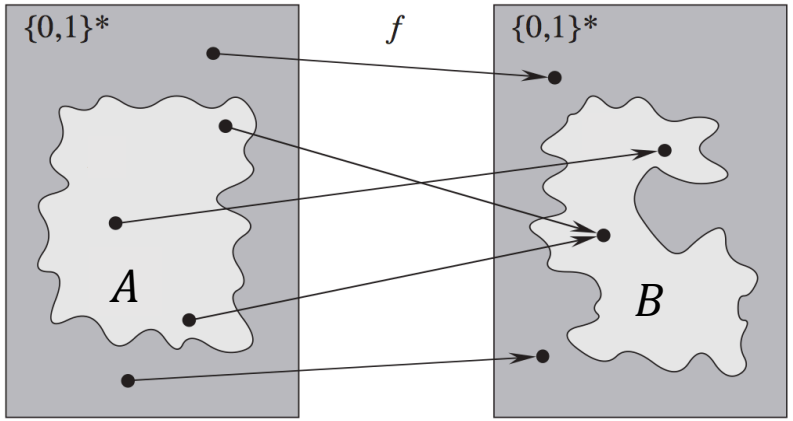
$AlgA$ outputs $YES$ iff
$AlgB$ outputs $YES$



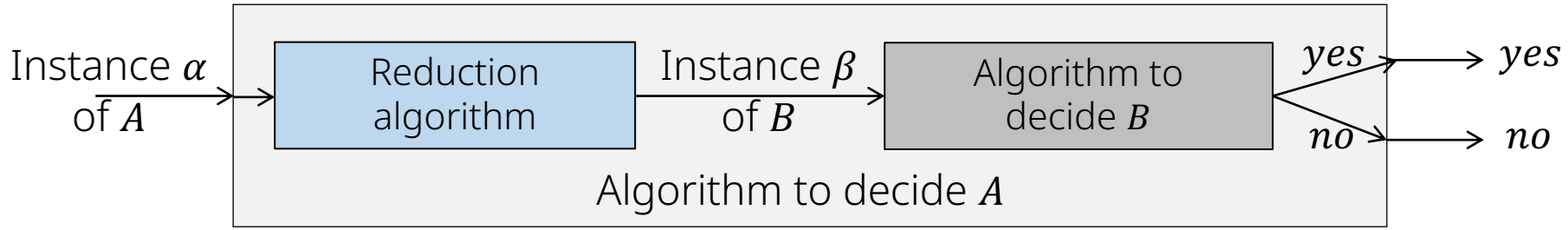
# Reduction

- To show the correctness of a p-time reduction:
  - Show that $B$ outputs 1 if and only if $A$ outputs 1
  - The reduction algorithm is in polynomial time



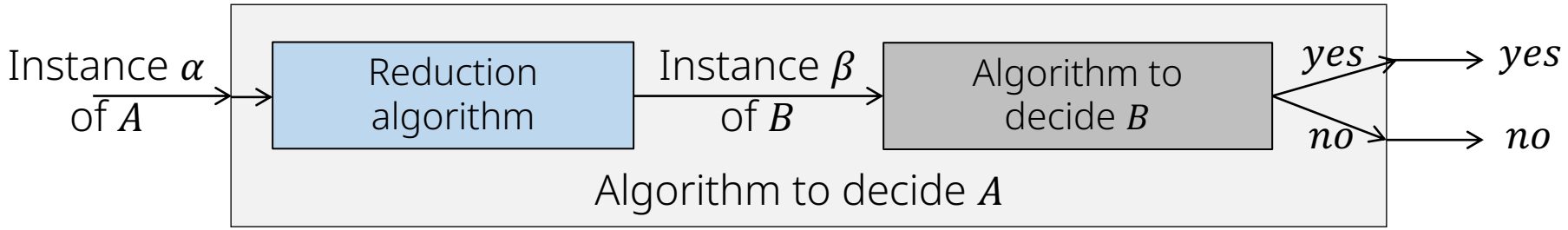



# Applications of reduction




1. **Designing algorithms**: given algorithm for $B$, we can also solve $A$
2. **Classifying problems**: establish relative difficulty between $A$ and $B$
3. **Proving limits**: if $A$ is hard, then so is $B$
   - Use frequently in NP-completeness proofs.
   - 目的是證明B為intractable，而非設計可用的演算法！

# Applications of reduction: designing algorithms



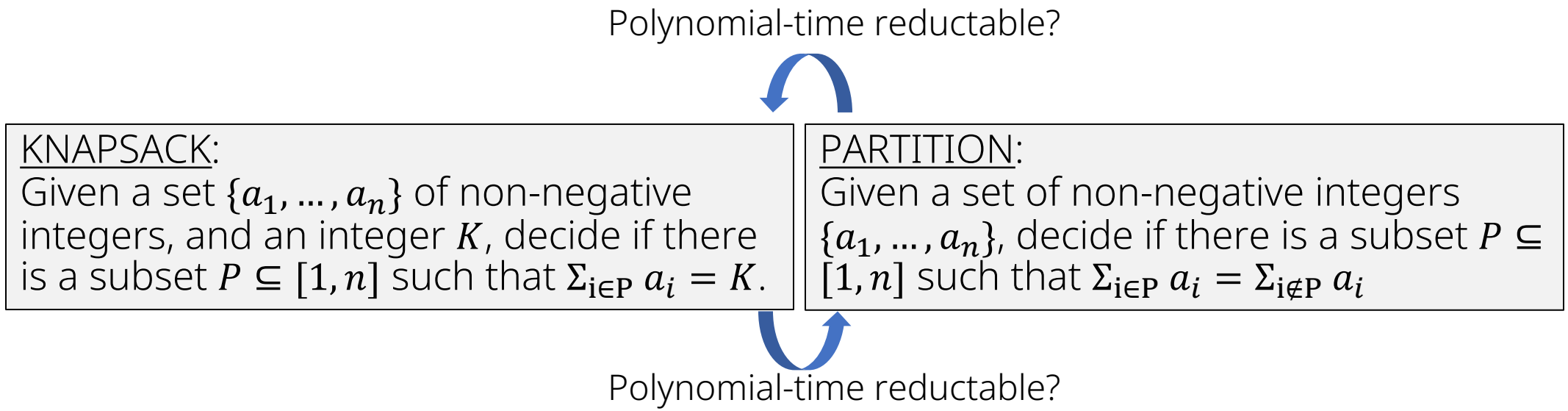

- Example #1:
  - A = all-pairs shortest path problem
  - B = single-source shortest path problem
- Example #2:
  - A = finding arbitrage opportunity
  - B = shortest path problem
- Any other examples?

# Applications of reduction: classifying problems

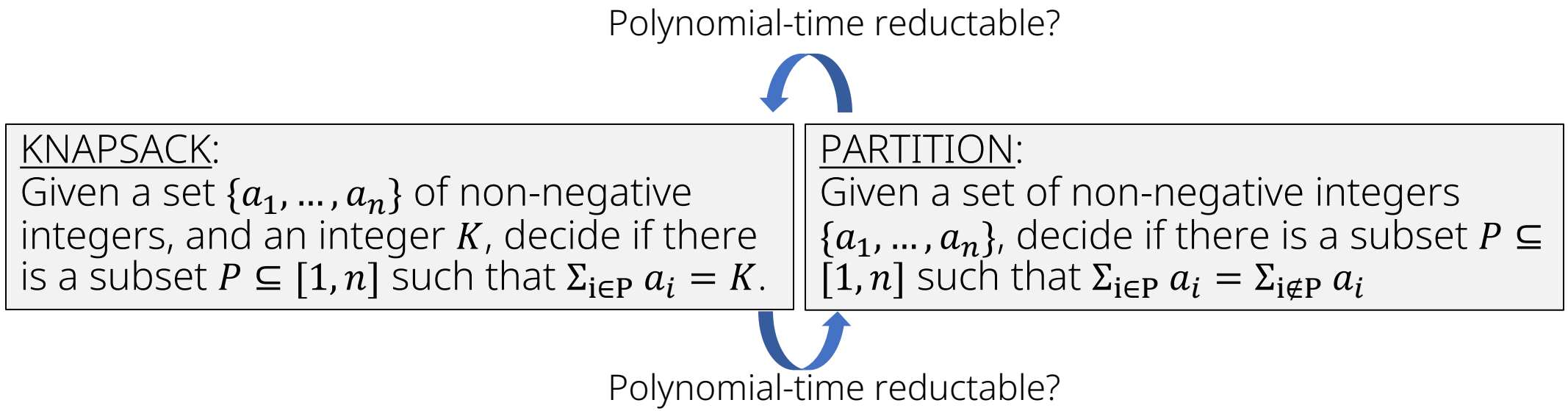
Goal: show KNAPSACK and PARTITION have equal difficulty

Approach: show that

- PARTITION $\leq_p$ KNAPSACK
- KNAPSACK $\leq_p$ PARTITION



# Polynomial-time reduction



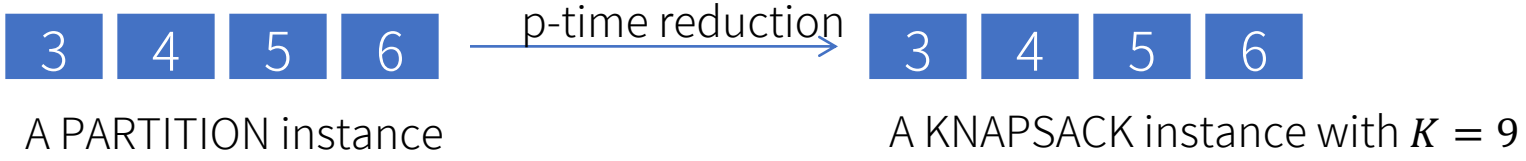

1. If we can solve KNAPSACK, how can we use that to solve PARTITION?
2. If we can solve PARTITION, how can we use that to solve KNAPSACK?

# 練習：證明 PARTITION $\leq_p$ KNAPSACK

KNAPSACK:
Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer $K$, decide if there is a subset $P \subseteq [1, n]$ such that $\Sigma_{i \in P}\, a_i = K$.

PARTITION:
Given a set of non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\Sigma_{i \in P}\, a_i = \Sigma_{i \notin P}\, a_i$

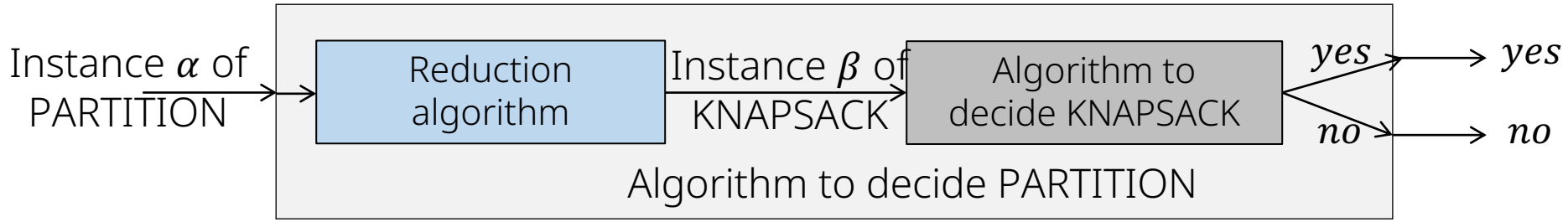
- If we can solve KNAPSACK, how can we use that to solve PARTITION?
- Polynomial-time reduction: Set $K = \frac{1}{2}\Sigma_{i \in 1,\ldots,n} a_i$




A PARTITION instance

A KNAPSACK instance with $K = 9$

# 練習：證明 PARTITION $\leq_p$ KNAPSACK




- If we can solve KNAPSACK, how can we use that to solve PARTITION?
- Polynomial-time reduction: Set $K = \frac{1}{2}\Sigma_{i\in 1,\ldots,n} a_i$
- Correctness proof: show that KNAPSACK returns yes if and only if an equal-size partition exists

# 練習：證明 KNAPSACK $\leq_p$ PARTITION

KNAPSACK:
Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer $K$, decide if there is a subset $P \subseteq [1, n]$ such that $\Sigma_{i \in P}\, a_i = K$.

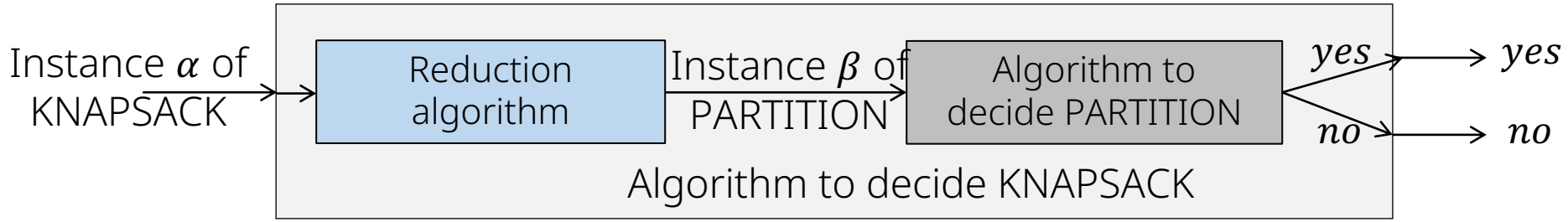
PARTITION:
Given a set of non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\Sigma_{i \in P}\, a_i = \Sigma_{i \notin P}\, a_i$

- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction:
  Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$, where $H = \frac{1}{2}\Sigma_{i \in 1,\ldots,n} a_i$


A KNAPSACK instance with K = 7

A PARTITION instance

# 練習：證明 KNAPSACK $\leq_p$ PARTITION




- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction: Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$, where $H = \frac{1}{2}\Sigma_{i\in 1,\ldots,n} a_i$
- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\Sigma_{i\in P}\, a_i = K$

"If" direction:


$= K$

$= (2H - K) + 2H + 2K$

$= 4H + K$

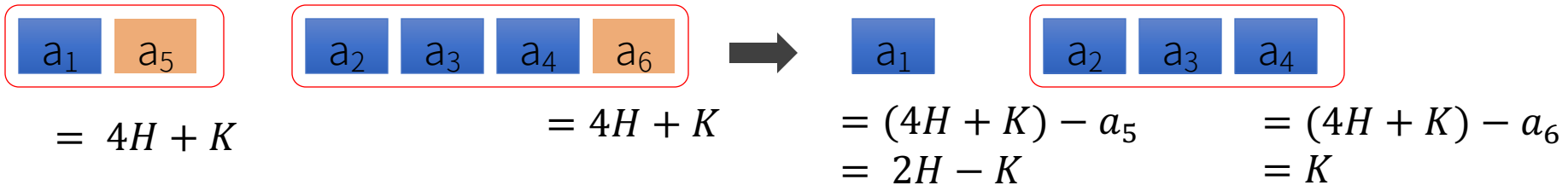
$= K + 4H$

# 練習：證明 KNAPSACK $\leq_p$ PARTITION

- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction: Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$, where $H = \frac{1}{2}\Sigma_{i\in 1,\ldots,n} a_i$
- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\Sigma_{i\in P}\, a_i = K$

"only if" direction:

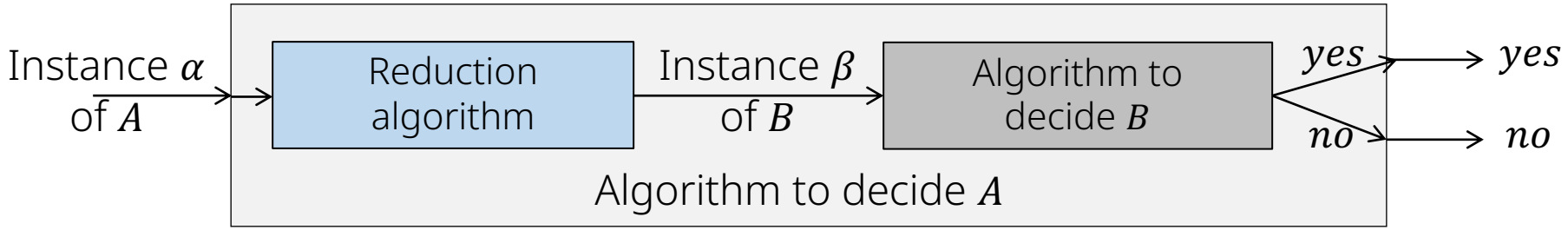
If PARTITION returns yes, $a_{n+1}$ & $a_{n+2}$ must be in different sets and each set = $4H + K$.
The two sets can only be equal if $\{a_1, \ldots, a_n\}$ are divided into $K$ and $2H - K$



# Applications of reduction: proving limits

- Reduction is useful to show one problem is no harder or no easier than the other one
  - We focus on the case in which both are decision problems
  - We will use it in almost every NP-completeness proof

# Applications of reduction: proving limits




- A common usage in NP-completeness proofs
  - Goal: prove that no known polynomial-time algorithm exists for $B$
  - Known: no known polynomial-time algorithm exists for $A$
  - Approach: construct a polynomial-time reduction algorithm to convert $\alpha$ to $\beta$
  - Why correct? 用反證法說明：若 $B$ 有p-time解，則根據 reduction，$A$ 也能在 p-time 解決。矛盾，故得証 $B$ 無 p-time 解。

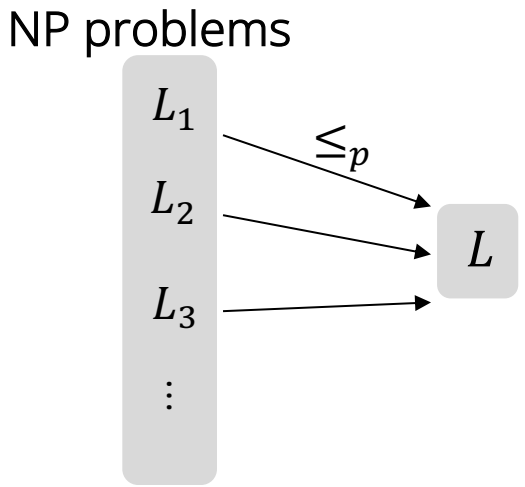
Q: 如果 reduction 不是 polynomial-time，這個證明還成立嗎？

# Proving NP-completeness

# Proving NP-Completeness

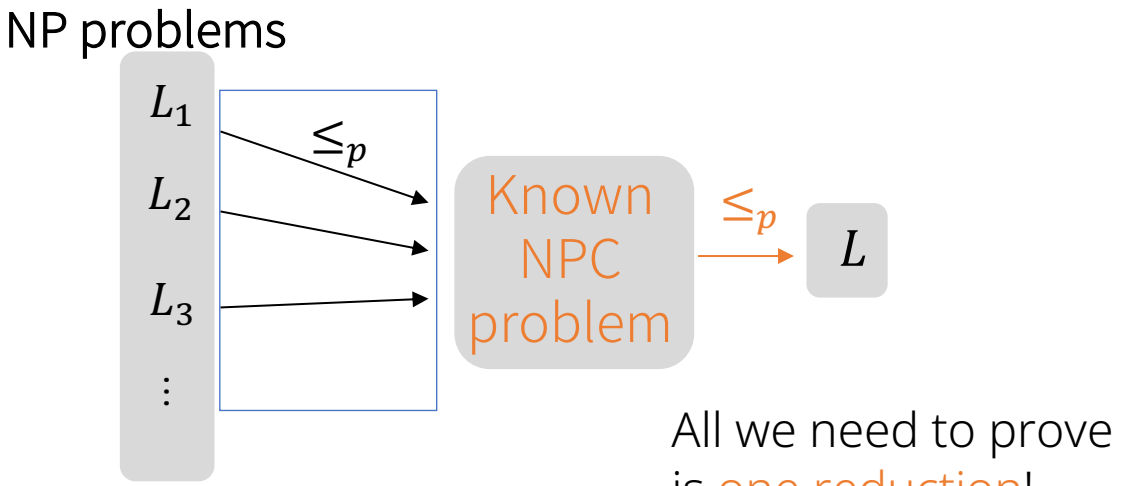
- Class NP-Complete (NPC): class of decision problems in both NP and NP-hard
- In other words, a decision problem $L$ is NP-complete if
  1. $L \in$ NP
  2. $L \in$ NP-Hard (that is, $L' \leq_p L$ for every $L' \in$ NP)

Q: How to prove NP-Hard (every NP problem reduces to L)?



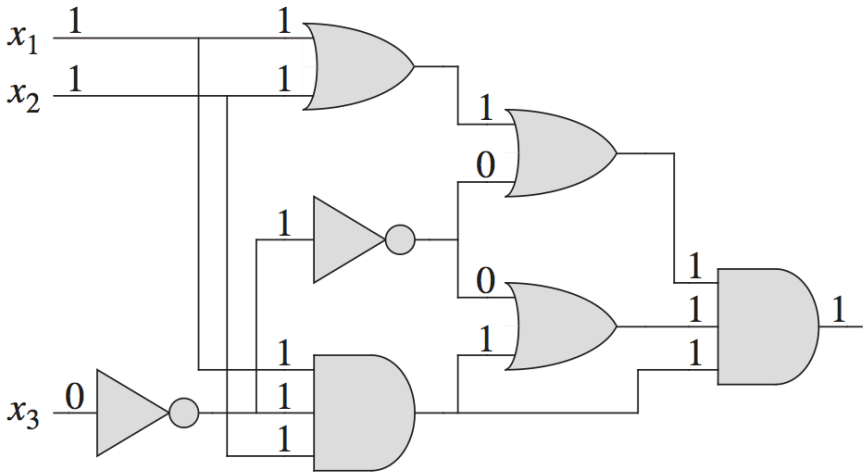
# Proving NP-Completeness

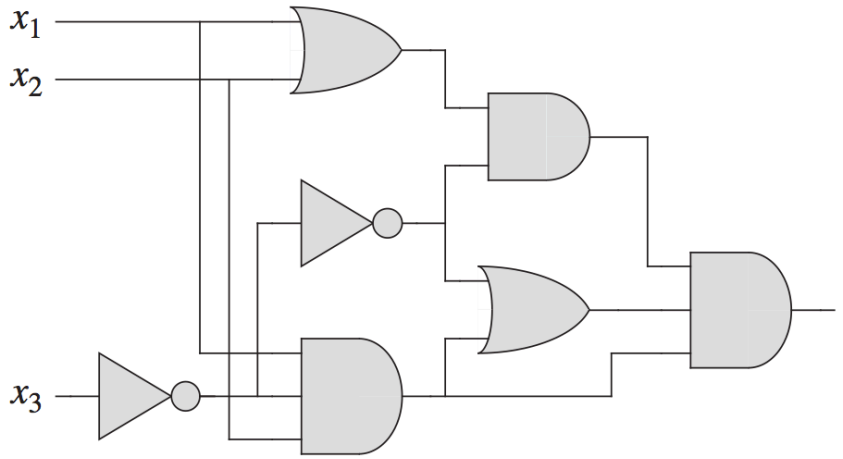
- Class NP-Complete (NPC): class of decision problems in both NP and NP-hard
- In other words, a decision problem $L$ is NP-complete if
  1. $L \in$ NP
  2. $L \in$ NP-Hard (that is, $L' \leq_p L$ for every $L' \in$ NP)



# The first NP-Complete problem

- Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
  - Satisfiable = there exists an assignment s.t. the circuit outputs 1



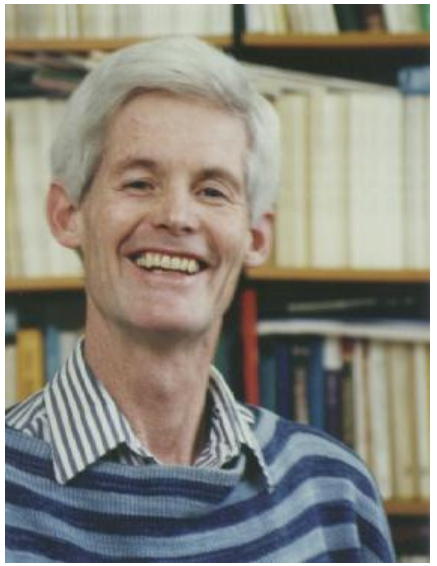



# Circuit-Satisfiability Problem

CIRCUIT-SAT = {<C>: C is a satisfiable Boolean combinational circuit}

**Cook's Theorem**

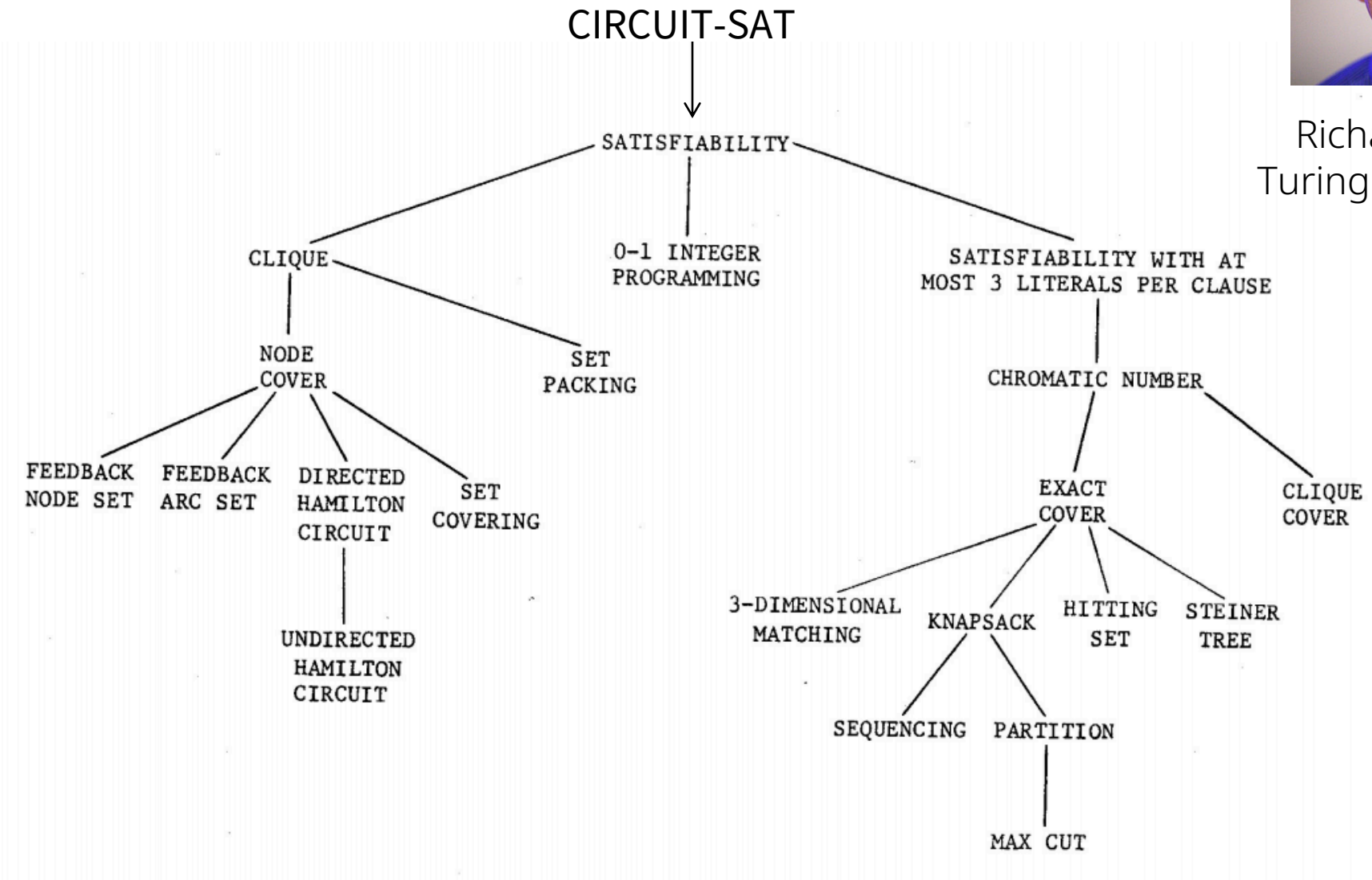
The Circuit-Satisfiability Problem (CIRCUIT-SAT) is NP-complete



Stephen Cook
Turing Award (1982)

# Karp's 21 NP-complete problems

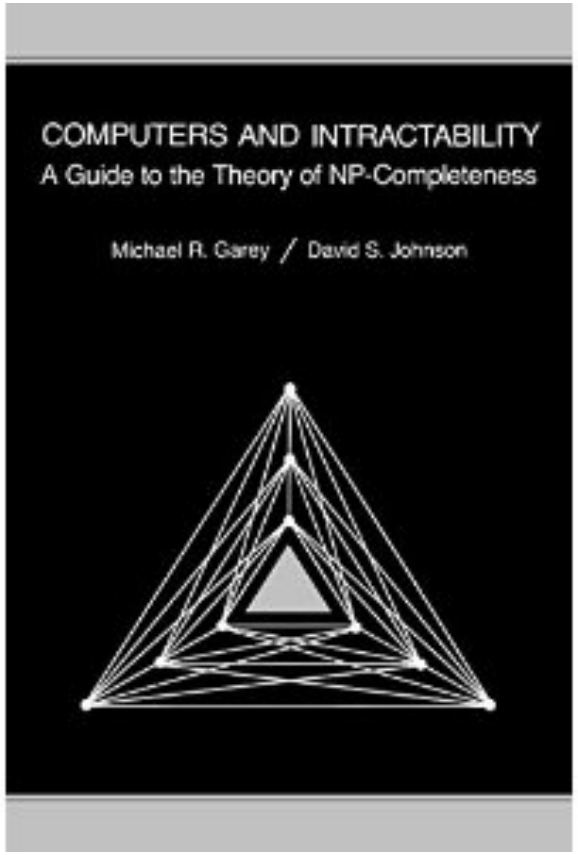
Richard M. Karp
Turing Award (1985)



# More NP-complete problems

- “Computers and Intractability” by Garey and Johnson includes more than 300 NP-complete problems
  - Most cited citations in CS

1. M R Garey, D S Johnson
   Computers and Intractability: A Guide to the Theory of NP-Completeness 1979
   8562
2. T H Cormen, C E Leiserson, R L Rivest
   Introduction to Algorithms 1991
   7126



# Proving NP-Completeness

- $L \in$ NP-Complete $\Leftrightarrow L \in$ NP and $L \in$ NP-hard
- Step-by-step approach for proving $L$ in NPC:

1. Prove $L \in$ NP
2. Prove $L \in$ NP-hard
   ① Select a known NPC problem $C$
   ② Construct a reduction $f$ transforming every instance o C to an instance of $L$
   ③ Prove that $x$ in $C$ if and only if $f(x)$ in $L$ for all $x$ in $\{0,1\}^*$
   ④ Prove that $f$ is a polynomial time transformation