INT102
Algorithmic Foundations And Problem Solving

The Limitations of Algorithm Power
-- Introduction to Computational Complexity Theory

Dr Jia Wang

Department of Intelligent Science

西交利物浦大学
Xi'an Jiaotong-Liverpool University

Acknowledgment: The slides are adapted from ones by Dr. Prudence Wong

What we have learnt so far

methodology / problems	Asym-ptotic idea	Brute force	Divide & Conquer	Dynamic Programming	Greedy	Space/Time	Branch&Bound	Backtracking
Efficiency	Big-O							
Sorting		Selection Bubble Insertion	Merge-sort			Count sorting		
Searching			Binary-searching					
String				LCS, Sequence Alignment		Horspool algorithm		
Graph		DFS/BFS		Floyed (All pair shortest path) Warshall (Transitive Closure) Assembly-line	Kruskal/Prim for MST Dikstra's For Shortest path		Traveling salesman, Job assignment	
Combinatory								n-Queens
Complexity	P/NP							

Introduction to Computational Complexity Theory ...

Agenda

- Warm up
  - Decision/Optimisation problems
  - Decision /Undecidable problems
  - Solving/Verifying a problem
    - Knapsack problem
    - Hamiltonian circuit problem
    - One more: Circuit-SAT

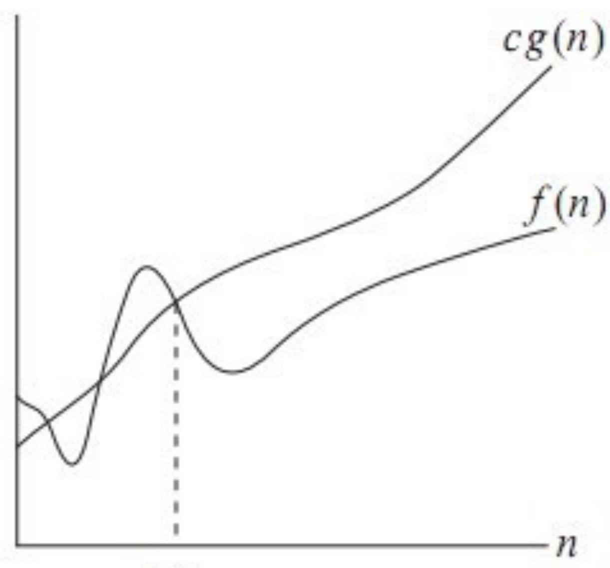
Complexity definitions (seen in 592)

- Big-Oh
- Big-Theta
- Big - Omega

Big Oh (O)

f(n)= O(g(n)) *iff* there exist positive constants c and n0 such that f(n) ≤ cg(n) for all n ≥ n0

O-notation to give an upper bound on a function

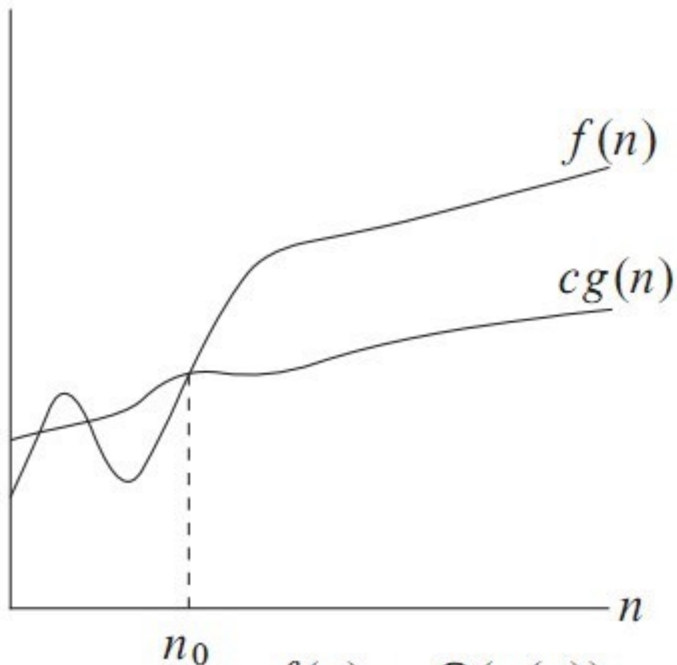


$f(n) = O(g(n))$

Omega Notation

Big oh provides an asymptotic upper bound on a function.
Omega provides an asymptotic lower bound on a function.

$$\Omega(g(n)) = \{ f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$

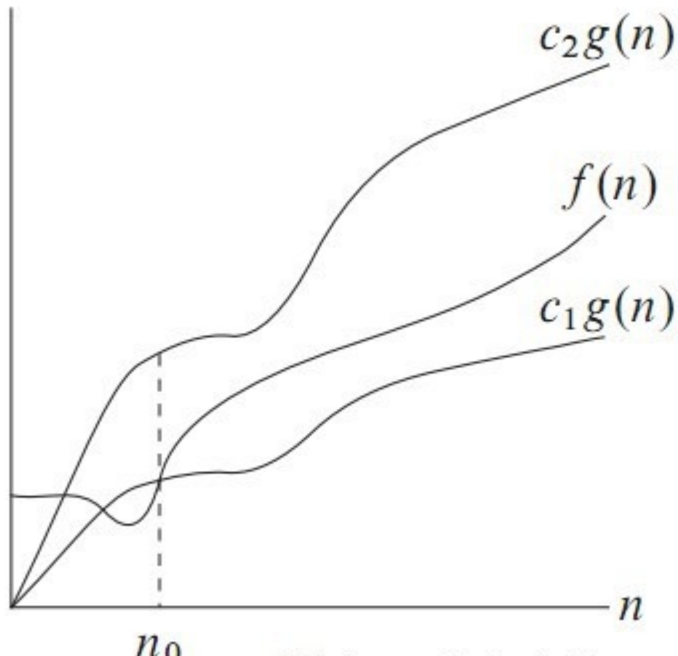


$f(n) = \Omega(g(n))$

Theta Notation

Theta notation is used when function *f* can be bounded both from above and below by the same function *g*

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}\ .^1$$



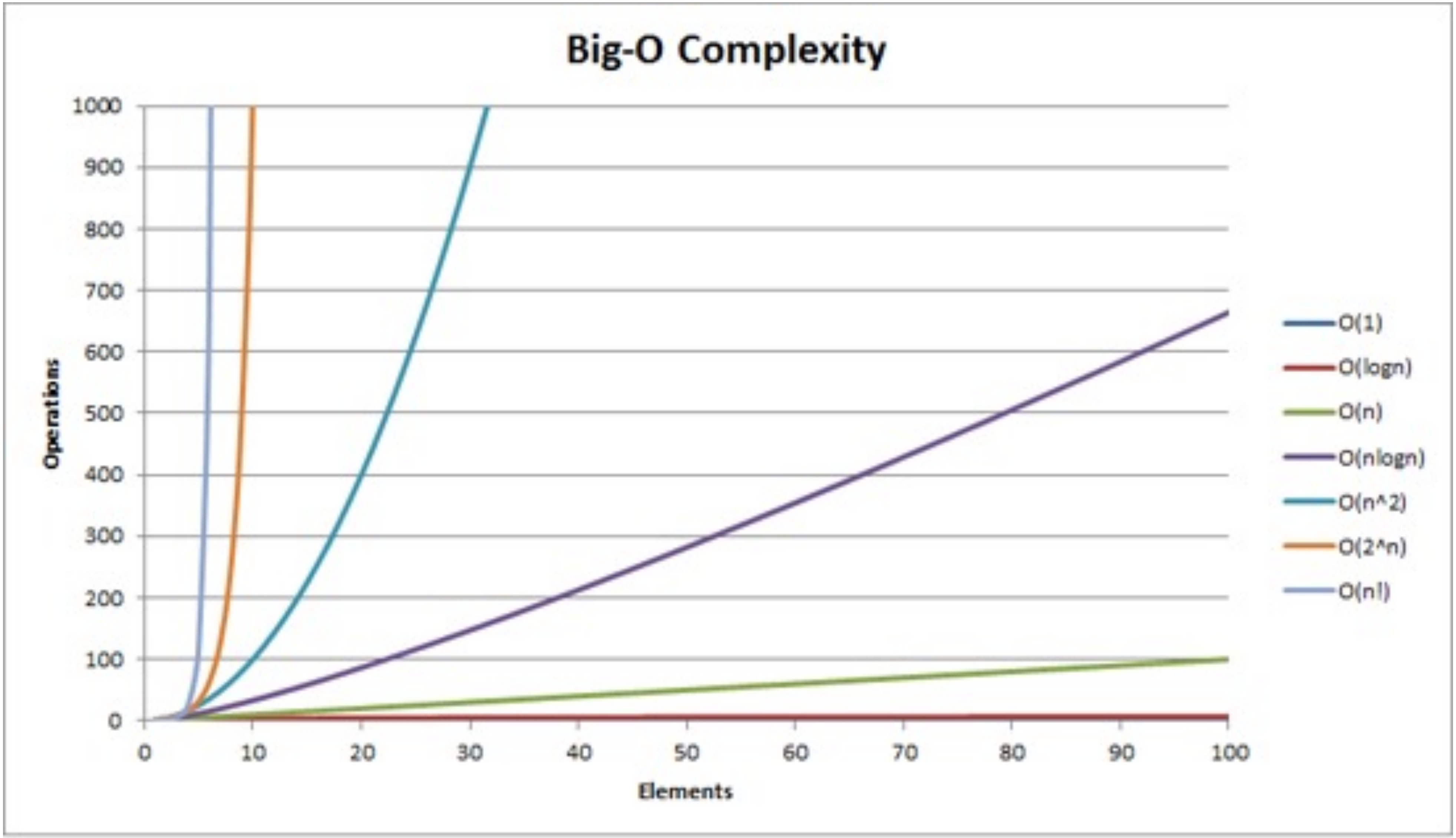
$f(n) = \Theta(g(n))$

Example: Bubble Sort

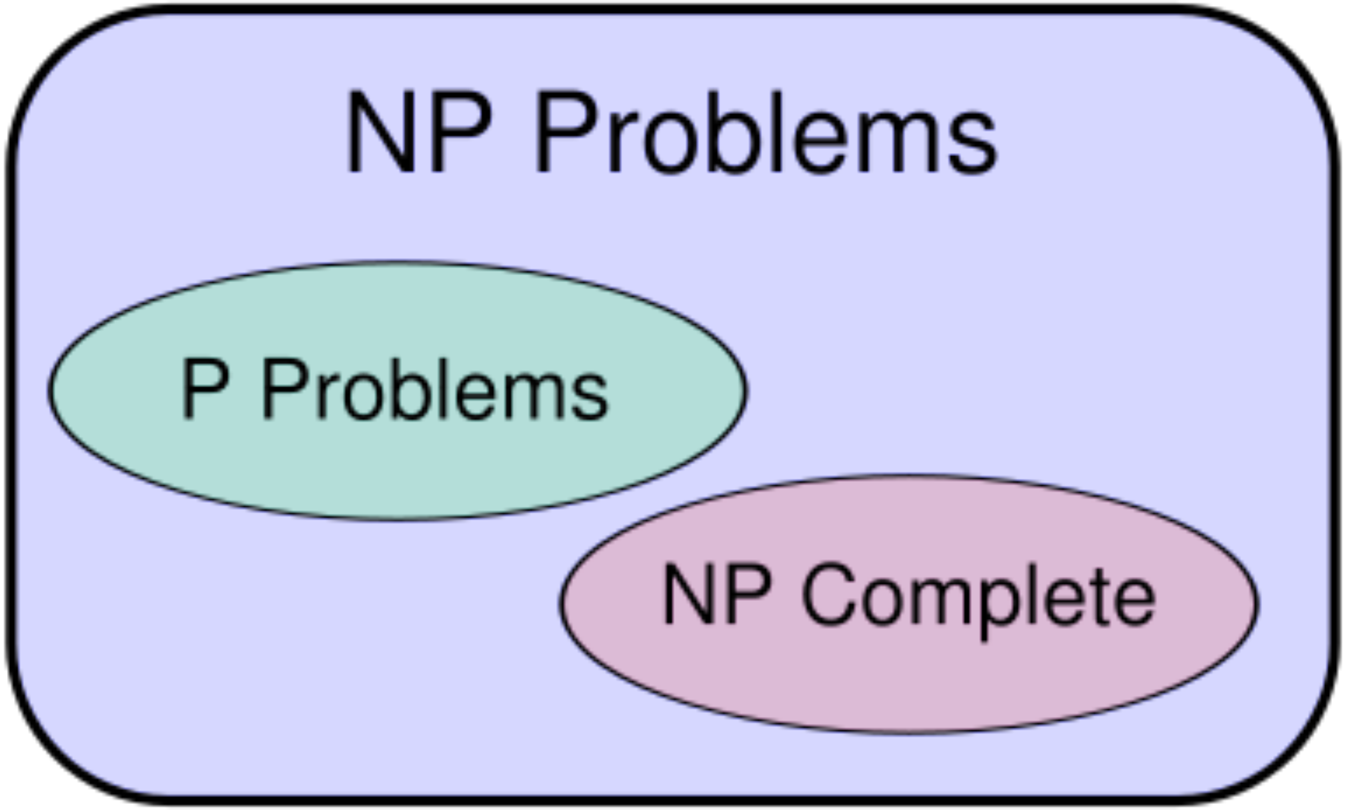
- **Big O (Worst Case):** $O(n^2)$. If the list is in reverse order, bubble sort will compare and swap elements $n(n-1)/2$ times, so the performance grows quadratically with the size of the input list, n.
- **Omega (Best Case):** $\Omega(n)$. If the list is already sorted, bubble sort only needs to pass through the list once, which involves $n-1$ comparisons and no swaps, making the performance grow linearly with the size of the input list, n.
- **Theta (Average Case):** $\Theta(n^2)$. While the best case is linear, the average and worst-case performance are typically quadratic, so bubble sort is often said to have a theta of n^2.

What we have learnt so far

- Polynomial Time
  - Linear Search O(n)
  - Binary Search O(logn)
  - Merge Sort O(nlogn)
  - Matrix MultiplicationO(n^3)
- Exponential Time
  - 0/1 Knapkack O(2^n)
  - Traveling SP O(2^n)
  - Hamilton circle O(2^n)


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Hard Computational Problems



Hard Computational Problems

An algorithm is efficient if its running time is bounded by a ***polynomial*** of its input size.

Some computational problems seems ***hard*** to solve.

Despite numerous attempts we ***do not know any efficient*** algorithms for these problems

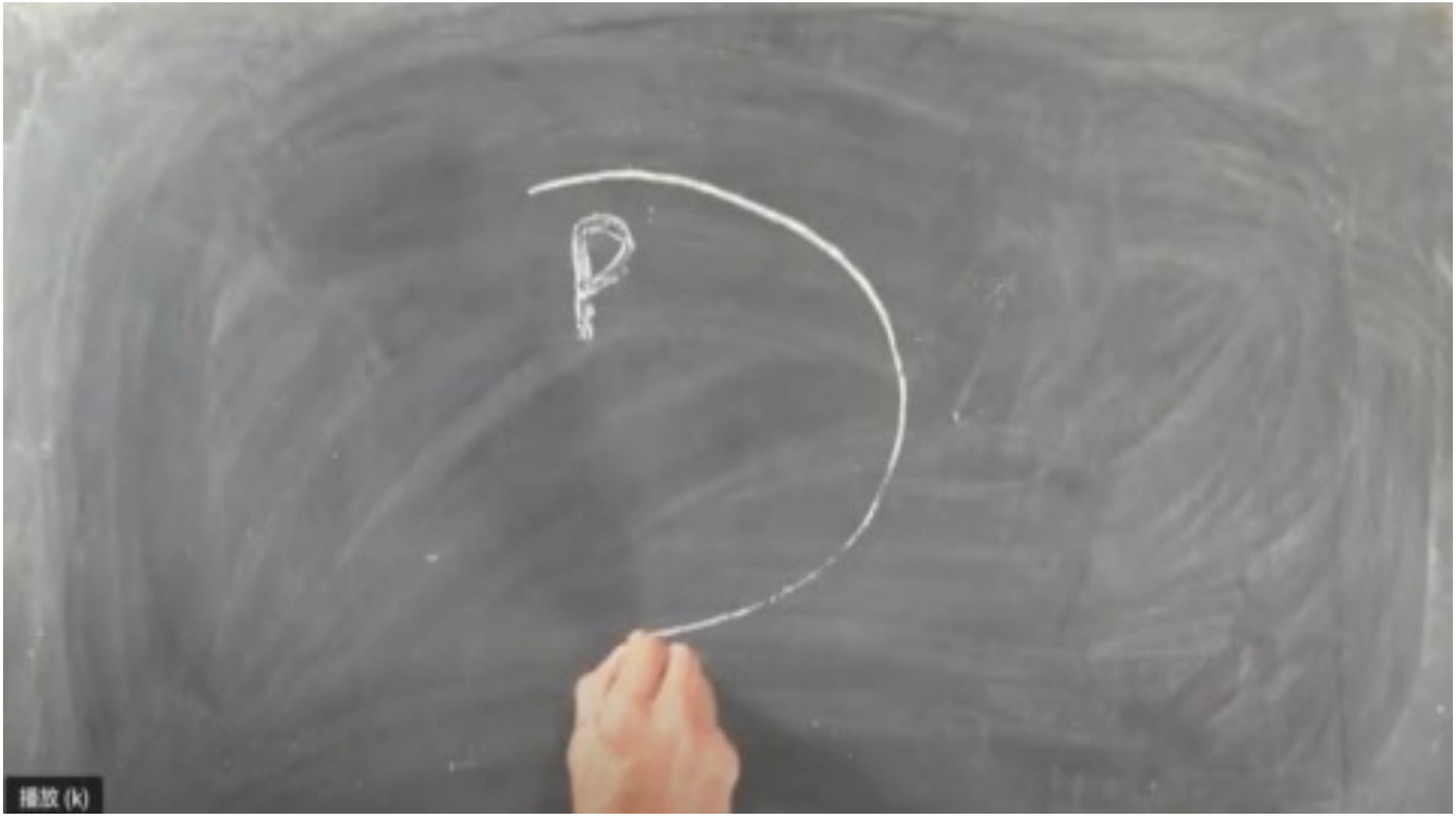
We are also far away from ***proving*** these problems are indeed hard to solve

In more formal language, we don't know whether *NP = P or NP ≠ P*. This is an important and fundamental question in theoretical computer science!

P = NP? http://www.claymath.org/millennium/P_vs_NP/

- Named as one of the seven "**Millennium Problems**" by the Clay Institute
- can earn you $1 million for its solution (and a place in mathematical and computer science history).
- Check www.claymath.org/millennium/ to read more about this (select the "P vs NP" link).

P vs. NP



A list of NP problems

- Hamiltonian Circuit
- Knapsack Problem
- Circuit-SAT

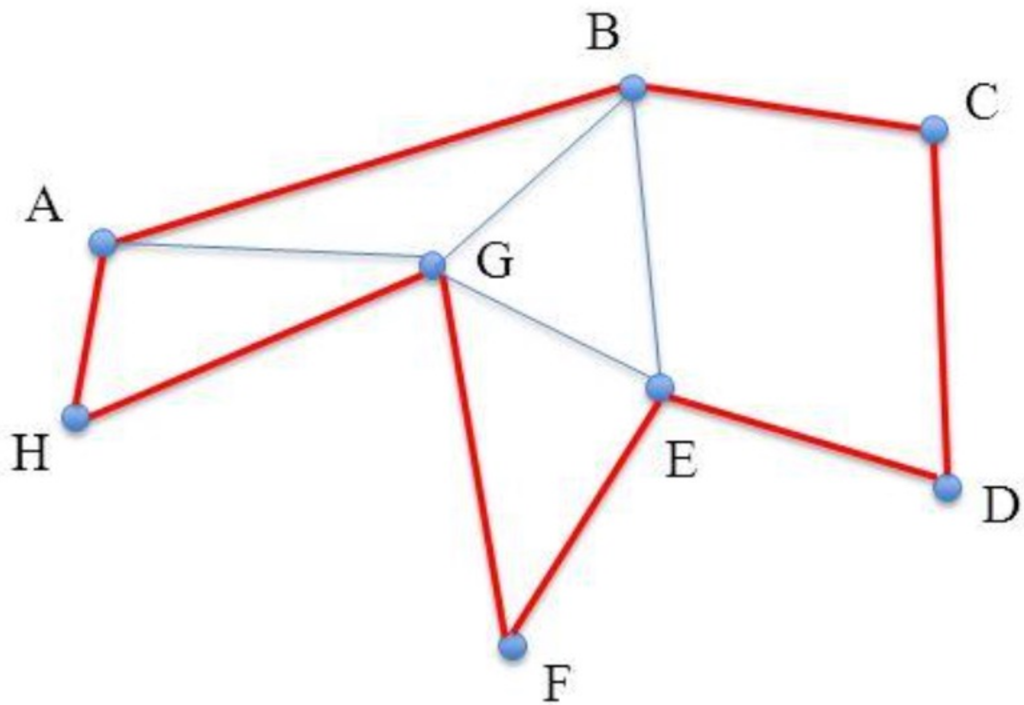
Hamiltonian Circuit

Input: A connected graph G

Question: Does G have a Hamiltonian circuit?
i.e., does G have a circuit that passes through every vertex exactly once, except for the starting and ending vertex?

Hamiltonian Circuit

A circuit that passes through every vertex at most once is called a **Hamiltonian circuit**.



A - H - G - F - E - D - C - B - A

Knapsack Problem

Input: Given n items with integer weights w_1, w_2, ..., w_n and integer values v_1, v_2, ..., v_n, and a knapsack with capacity W.

Problem (optimisation version): Find a subset of items whose total weight does not exceed W and that maximises the total value.

(taking fractional parts of items is NOT allowed)

Known as 0/1 Knapsack Problem

Knapsack Problem

n = 4,
W = 50

Boolean Circuit (Circuit-SAT Preparation)

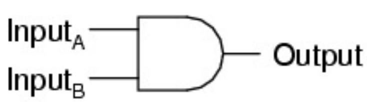
In discrete Math, you learned about *propositional logic*, and the basic operations for combining truth values.

A **Boolean Circuit** is a directed graph where each vertex, called a *logic gate* corresponds to a simple Boolean function, one of **AND, OR, or NOT**.

Incoming edges: inputs for its Boolean function

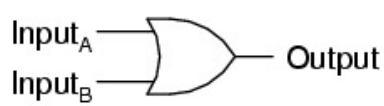
Outgoing edges: outputs

2-input AND gate



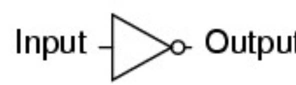
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

2-input OR gate



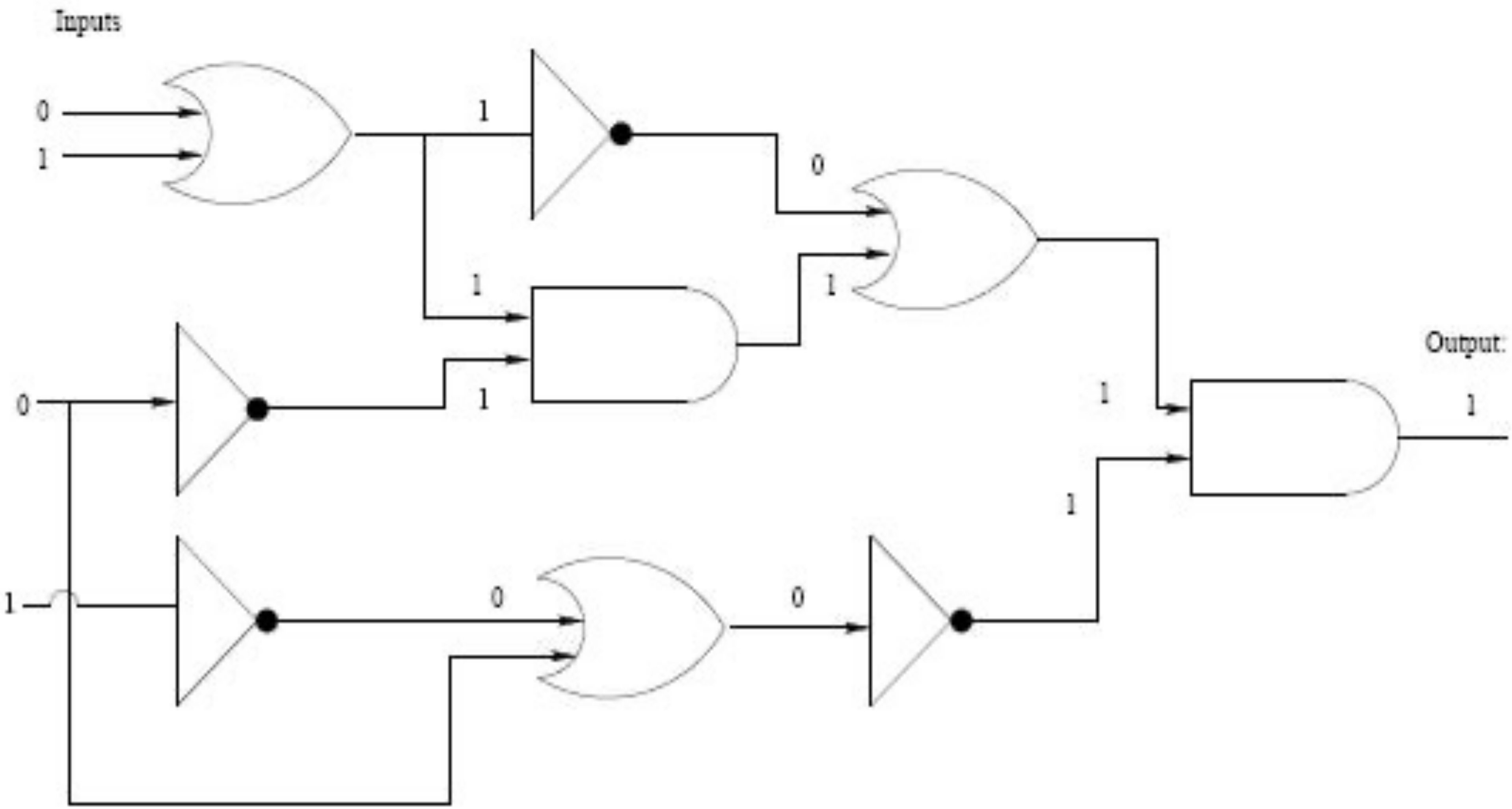
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT gate truth table



Input	Output
0	1
1	0

Boolean Circuit - Example



Examples

- We have seen two examples of these difficult problems where no efficient (polynomial) algorithm is known
  - Knapsack problem
  - Hamiltonian circuit problem
  - One more: Circuit-SAT

All we know is to exhaust all possible solutions to find the best one

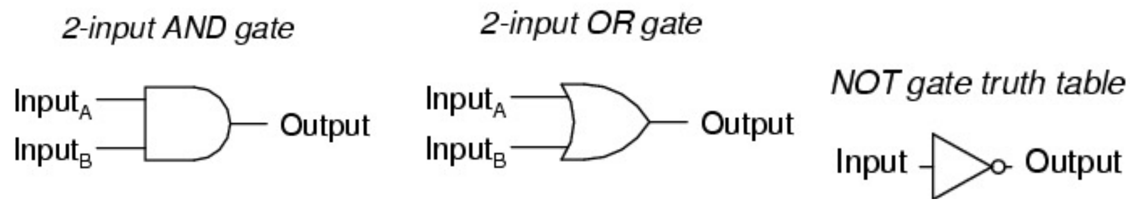
Circuit-SAT

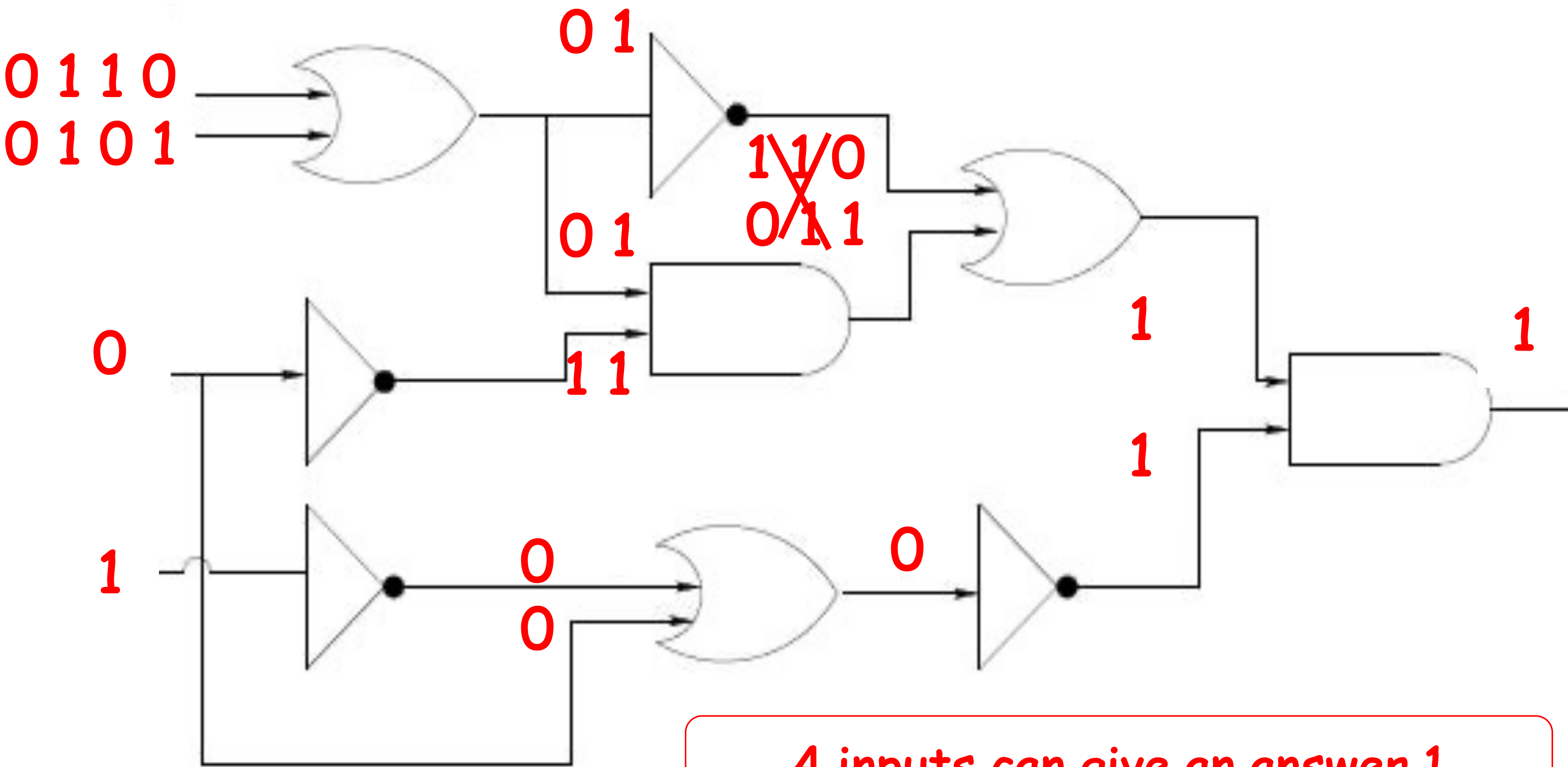
Input: a Boolean Circuit with a single output vertex

Question: is there an assignment of values to the inputs so that the output value is 1?

SAT means **satisfiability**

Circuit-SAT





4 inputs can give an answer 1

why study NP/P?

1. **Understanding Computational Limits**: Knowing whether a problem is in P or NP helps understand the inherent difficulty of solving certain problems and whether they can be feasibly solved as the size of the input grows.

2. **Algorithm Development**: If a problem is in P, it can usually be solved efficiently, which encourages the development of effective algorithms. If it's NP-complete or NP-hard, while a polynomial-time solution may not be currently available, it motivates the search for approximate algorithms or heuristics that can solve the problem reasonably well in practice.

3. **P vs NP Question**: This is one of the seven Millennium Prize Problems for which the Clay Mathematics Institute offers a $1 million prize for a correct solution. Determining whether P equals NP is fundamental to understanding the limits of what can be computed.

Any more Problems are NP-hard?

Decision/Optimisation problems

A ***decision*** problem is a computational problem for which the output is either ***yes*** or ***no***.

In an ***optimisation*** problem, we try to ***maximise*** or ***minimise*** some value.

An optimisation problem can be ***turned into*** a decision problem if we add a ***parameter k***, and then ask whether the optimal value in the optimisation problem is ***at most*** or ***at least*** k.

Note that if a decision problem is ***hard***, then its related optimisation version must also be ***hard***.

Example - MST

Optimisation problem: Given a graph G with integer weights on its edges. What is the weight of a *minimum* spanning tree (MST) in G?

Decision problem: Given a graph G with integer weights on its edges, and *an integer k*. Does G have a MST of weight *at most k*?

Example - Knapsack problem

Input: Given n items with integer weights w_1, w_2, ..., w_n and integer values v_1, v_2, ..., v_n, a knapsack with capacity W and a value k.

Optimisation problem: Find a subset of items whose total weight does not exceed W and that *maximises* the total value.

Decision problem: For *any integer k*, is there a subset of items whose total weight does not exceed W and whose total value is *at least k*?

Exercise

State the decision version of the following problems

- Given a **weighted** graph G and a source vertex a, find the ***shortest*** paths from a to every other vertex

Exercise - Solution

- Given a weighted graph G, a source vertex a and ***a value k***, are there shortest paths from a to every other vertex such that each path is of weight ***at most k***?

How can solving a decision problem helps to find solution to the optimization problem?

Can All Decision Problems Be Solved By Algorithms?

- The Answer is No.
- The problems can not solved by algorithms is called undecidable problems.
- One such a problem is Halting Problem (AlanTuring 1936)

  "Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it."

Proof: Short but not easy to understand.

Sketch of proof on Halting Problem

If there is an algorithm *A* such that for any program *P* and input *I*

$$A(P, I) = \begin{cases} 1 & \text{if } P \text{ halts on input I} \\ 0 & \text{if } P \text{ does not halt on input } I \end{cases}$$

Consider the following program Q

$$Q(P) = \begin{cases} \textit{Halts}, & \text{if } A(P, P) = 0, \textit{ ie. P does not halt on input P} \\ \textit{Does not halt}, & \text{if } A(P,P) = 1, \textit{ ie. P does halt on input P} \end{cases}$$

Now replace *P* by *Q* in *Q(P)*

$$Q(Q) = \begin{cases} \textit{Halts}, & \text{if } A(Q, Q) = 0, \textit{ ie. Q does not halt on input Q} \\ \textit{Does not halt}, & \text{if } A(Q,Q) = 1, \textit{ ie. Q does halt on input Q} \end{cases}$$

Solving/Verifying a problem

Solving a problem is different from verifying a problem

- solving: we are given an input, and then we have to FIND the solution
- verifying: in addition to the input, we are given a "certificate" and we verify whether the certificate is indeed a solution

We may not know how to solve a problem efficiently, but we may know how to verify whether a candidate is actually a solution

Example - Hamiltonian circuit problem

Suppose through some (unspecified) means (like good guessing), we find a *candidate* for a Hamiltonian circuit, i.e. a list of vertices and edges that ***might be a Hamiltonian circuit*** in the input graph G.

It is easy to check if this is indeed a Hamiltonian circuit. Check

- that all the proposed edges exist in G,
- that we indeed have a cycle, and
- that we hit every vertex in G once.

If the candidate solution is indeed a Hamiltonian circuit, then it is a ***certificate*** verifying that the answer to the decision problem is "Yes"

Example – 0/1 Knapsack Problem

Consider an instance of the 0/1 Knapsack problem (decision version)

Suppose someone proposes a subset of items, it is easy to check

- if those items have **total weight at most W** and
- if the **total value is at least k**

- If both conditions are true, then the subset of items is a ***certificate*** for the decision problem
  - i.e., it verifies that the answer to the 0/1 knapsack decision problem is "Yes"

Example – Circuit-SAT

Consider a Boolean Circuit

Suppose someone proposes an assignment of truth values to the input, it is easy to check

- if the input values lead to a final value of 1 in the output
- this is done by checking every logic gate

If the input truth values give a final value of 1, these values form a ***certificate*** for the decision problem