

# *Algorithm Design and Analysis*

---

**CSE  
565**

## **LECTURE 35**

**Computational**

**Intractability**

- Polynomial Time  
Reductions

**Adam Smith**

## This chapter: computational intractability and NP

- Goal: understand what “cannot” be computed
  - can you name some “hard” problems?
- Computability: “cannot, even in principle”
- Complexity: “cannot given reasonable resources”
  - we focus on complexity here (why?)
- Central ideas:
  - poly-time as “feasible”
  - most natural problems are either easy or have no known poly-time algorithms
  - NP and NP-completeness
  - P = problems that are easy to answer
    - e.g. minimum cut
  - NP = {problems whose answers are easy to **verify** given **hint**}
    - e.g. graph 3-coloring
  - NP-completeness
  - many **natural** problems are easy if **and only if** P=NP

“unreasonable effectiveness of mathematics in science”

So far: understand alg. design

- Greedy.
- Divide-and-conquer.
- Dynamic programming.
- Duality (e.g. MaxFlow-MinCut)
- Reductions.
- Local search.
- Randomization.
- ...

Examples.

$O(n \log n)$  interval scheduling.

$O(n \log n)$  FFT.

$O(n^2)$  edit distance.

$O(n^3)$  bipartite matching.

New goal: understand what is hard to compute

- NP-completeness.
- PSPACE-completeness.
- Undecidability.

$O(n^k)$  algorithm unlikely.

$O(n^k)$  certification algorithm unlikely.

No algorithm possible.

# Example hard problems?

Problems in P	Problems in NP, not known to be in P or NP-complete	NP-complete	PSPACE-complete	Undecidable
max flow, knapsack with fractional elements in knapsack	factoring, graph isomorphism	longest path, traveling salesman, graph coloring, 3-sat, independent set, knapsack w/ fract. weights	TQBF (totally quantified boolean formula), Given a game (with poly-size board configurations), decide if white has winning strategy	halting problem

# Understanding "hardness"

## Computability

- Understand what can/cannot be computed, even in principle
- "Halting problem" : given program code, will this program get stuck in an infinite loop?
  - **Theorem** (Turing): "halting problem" is **uncomputable**
  - ... so most things we want compilers to do are undecidable
- **Corollary**: Some numbers are too big to compute
  - "busy beaver problem":  $BB(n) = \max$  # of steps that an **n-character** C program (that stops eventually) can take before it stops.
  - If you could compute an upper bound on  $BB(n)$ , you could solve the halting problem
  - (Competition: think of the biggest number you can!)


## Complexity

- Understand what problems cannot be solved **in a reasonable amount of time**
- Far less is known! Famous problem: P vs. NP.

## Central ideas we'll cover

- Poly-time as "feasible"
- most natural problems **either** are easy (say  $n^3$ ) **or** have no known poly-time algorithms
- Reductions: X is no harder than Y
- P = problems that are easy to answer
  - e.g. minimum cut
- NP = {problems whose answers are easy to **verify** given **hint**}
- e.g. graph 3-coloring
- NP-completeness
  - many **natural** problems are easy **if and only if** P=NP

"unreasonable effectiveness of mathematics in science"



# What do we mean by "feasible"?

Q. Which problems will we be able to solve in practice?

A working definition. [von Neumann 1953, Godel 1956, Cobham 1964, Edmonds 1965, Rabin 1966]  
Those with polynomial-time algorithms.

Yes	Probably no
Shortest path	Longest path
Matching	3D-matching
Min cut	Max cut
2-SAT	3-SAT
Planar 4-color	Planar 3-color
Bipartite vertex cover	Vertex cover
Primality testing	Factoring

# Classify Problems

Desiderata. Classify problems as those that can be solved in polynomial-time and those that cannot.

Roughly: C program on machine with infinite memory

Provably require exponential-time:

- Given a **Turing machine**, does it halt in at most  $k$  steps?
- Given a board position in an  $n$ -by- $n$  generalization of chess, can black guarantee a win?

Frustrating news. Huge number of fundamental problems have defied classification for decades.

This chapter. Show that these fundamental problems are "computationally equivalent" and appear to be different manifestations of one **really hard** problem.

## Tool: Polynomial-Time Reduction

Desiderata'. Suppose we could solve  $X$  in polynomial-time. What else could we solve in polynomial time?

don't confuse with reduces from

Reduction. Problem  $X$  **polynomially reduces to** problem  $Y$  if arbitrary instances of problem  $X$  can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem  $Y$ .

Notation.  $X \leq_{P, \text{Cook}} Y$  (or  $X \leq_P Y$ ).

computational model supplemented by special piece of hardware that solves instances of  $Y$  in a single step

Later in the lecture.  $X \leq_{P, \text{Karp}} Y$ .

Remarks.

- We pay for time to write down instances sent to black box  $\Rightarrow$  instances of  $Y$  must be of polynomial size.

# Polynomial-Time Reduction

Purpose. Classify problems according to **relative** difficulty.

Design algorithms. If  $X \leq_p Y$  and  $Y$  can be solved in polynomial-time, then  $X$  can also be solved in polynomial time.

Establish intractability. If  $X \leq_p Y$  and  $X$  cannot be solved in polynomial-time, then  $Y$  cannot be solved in polynomial time.

Establish equivalence. If  $X \leq_p Y$  and  $Y \leq_p X$ , we use notation  $X \equiv_p Y$ .

  
up to cost of reduction

## Simplifying Assumption: Decision Problems

Search problem. Find some structure.

Example. Find a minimum cut.

Decision problem.

- $X$  is a set of strings.
- Instance: string  $s$ .
- If  $x \in X$ ,  $x$  is a YES instance; if  $x \notin X$  is a NO instance.
- Algorithm  $A$  solves problem  $X$ :  $A(s) = \text{yes}$  iff  $s \in X$ .

Example. Does there exist a cut of size  $\leq k$ ?

Self-reducibility. Search problem  $\leq_{P, \text{Cook}}$  decision version.

- Applies to all (NP-complete) problems in this chapter.
- Justifies our focus on decision problems.

# Polynomial Transformation

Def. Problem  $X$  **polynomial reduces** (Cook) to problem  $Y$  if arbitrary instances of problem  $X$  can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem  $Y$ .

Def. Problem  $X$  **polynomial transforms** (Karp) to problem  $Y$  if given any input  $x$  to  $X$ , we can construct an input  $y$  such that  $x$  is a  $_{yes}$  instance of  $X$  iff  $y$  is a  $_{yes}$  instance of  $Y$ .

↑  
we require  $|y|$  to be of size polynomial in  $|x|$

Note. Polynomial transformation is polynomial reduction with just one call to oracle for  $Y$ , exactly at the end of the algorithm for  $X$ .

Open question. Are these two concepts the same?

↑  
Caution: KT abuses notation  $\leq_p$  and blurs distinction