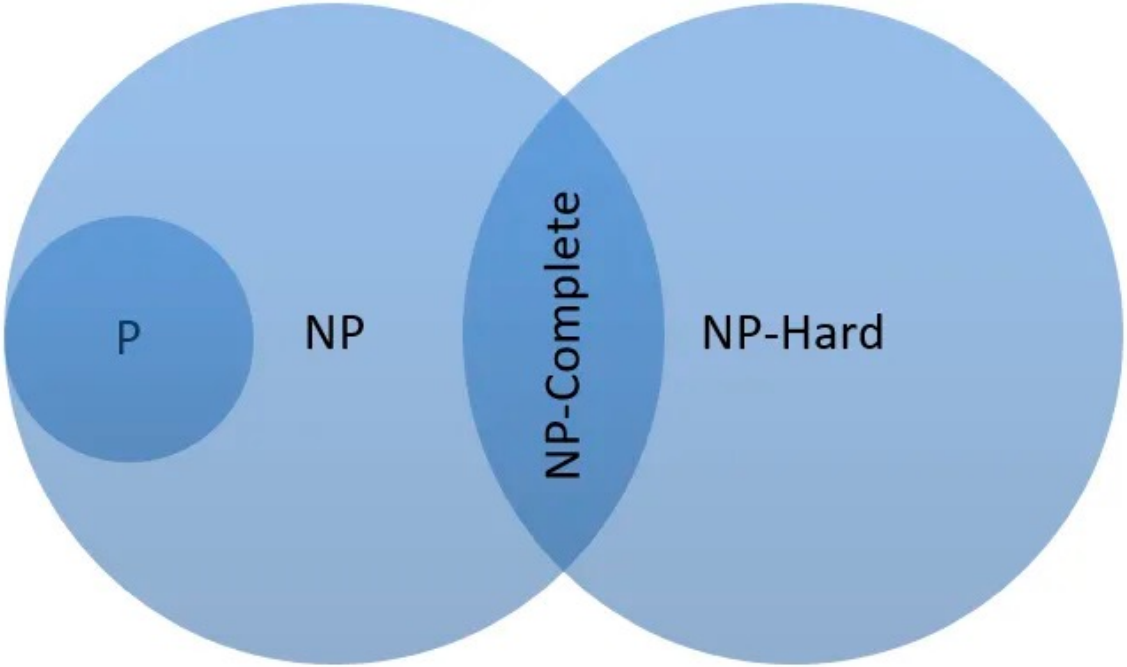
NP-Completeness



NYIT CSCI-651

Complexity Recap

– constant:	$O(1)$	
– logarithmic:	$O(\log n)$	($\log_k n$, $\log n^2 \in O(\log n)$)
– poly-log:	$O(\log^k n)$	(k is a constant >1)
– linear:	$O(n)$	
– (log-linear):	$O(n \log n)$	(usually called "n log n")
– (superlinear):	$O(n^{1+c})$	(c is a constant, $0 < c < 1$)
– quadratic:	$O(n^2)$	
– cubic:	$O(n^3)$	
– polynomial:	$O(n^k)$	(k is a constant) "tractable"
– exponential:	$O(c^n)$	(c is a constant > 1)

"intractable"

51

Tractability

- Some problems are *intractable*:
  as they grow large, we are unable to solve them in reasonable time
- What constitutes reasonable time?
  - Standard working definition: *polynomial time*
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k

- Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Polynomial-Time Algorithms

- Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

Optimization/Decision Problems

- Optimization Problems
  - An optimization problem is one which asks: "What is the optimal solution to problem X?"
  - Examples:
    - Knapsack problem
    - Minimum Spanning Tree
- Decision Problems
  - A decision problem is one with yes/no answer
  - Examples:
    - Does a graph G have a MST of weight $\leq$ W?

Optimization/Decision Problems

- An optimization problem tries to find an optimal solution
- A decision problem tries to answer a yes/no question
- Many problems will have decision and optimization versions
  - Eg: Traveling salesman problem
    - optimization: find Hamiltonian cycle of minimum weight
    - decision: is there a Hamiltonian cycle of weight $\leq$ k

A Hamiltonian cycle (or Hamiltonian circuit) is a cycle in an undirected graph that visits each vertex exactly once and also returns to the starting vertex.

Optimization/Decision Problems

- Some problems are decidable, but *intractable*:

  as they grow large, we are unable to solve them in reasonable time

- *Is there a polynomial-time algorithm that solves the problem?*

The Class *P*

P: the class of decision problems that have polynomial-time deterministic algorithms.

- they are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n
- A deterministic algorithm is (essentially) one that always computes the same correct answer

Why polynomial?

- if not, very inefficient
- nice closure properties
  - *the sum and composition of two polynomials are always polynomials too*

Sample Problems in P

- Fractional Knapsack
- MST
- Sorting

The class *NP*

NP: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a nondeterministic algorithm)

- A *deterministic* computer is what we know
- A *nondeterministic* computer is one that can "guess" the right answer or solution
- Think of a nondeterministic computer as a parallel machine that can freely spawn ***an infinite number*** of processes

- Thus *NP* can also be thought of as the class of problems whose solutions can be verified in polynomial time

- Note that *NP* stands for "Nondeterministic Polynomial-time"

- **P**: The class of *problems* for which polynomial time ($O(n^k)$ for some constant **k**) algorithms exist (to solve the problem)

- **NP**: The class of *problems* for which polynomial time algorithms exist to check that an answer is "yes"

- There are many important problems for which:
  - We know they are in **NP**
  - We do not know if they are in **P** (but we *highly* doubt it)
  - The best algorithms we have are exponential
    - $O(\mathbf{k}^{\mathbf{n}})$ for some constant **k**

Sample Problems in NP

Subset sum

31?

Input: An *array* of *n* numbers and a target-sum *sum*

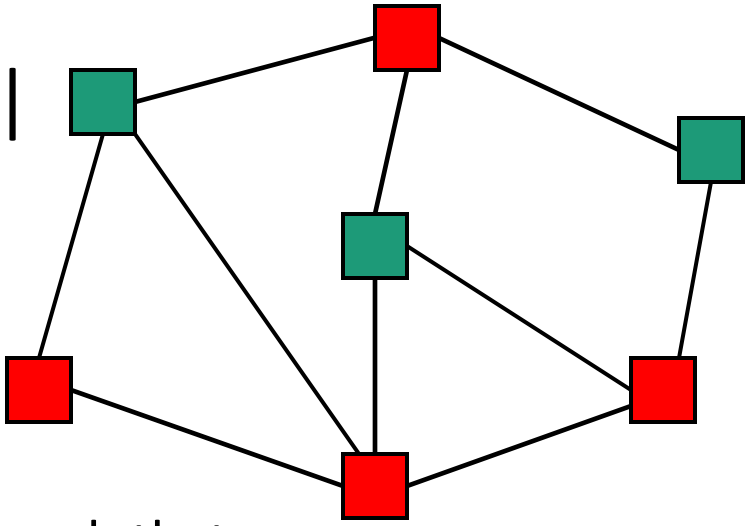
Output: A subset of the numbers that add up to *sum* if one exists

$O(2^n)$ algorithm: Try every subset of array

$O(n^k)$ algorithm: Unknown, probably does not exist

Verifying a solution: Given a subset that allegedly adds up to sum, add them up in $O(n)$

Vertex Cover: Optimal



Input: A graph (**V**,**E**)

Output: A minimum size subset **S** of **V** such that

for every edge (**u**,**v**) in **E**, at least one of **u** or **v** is in **S**

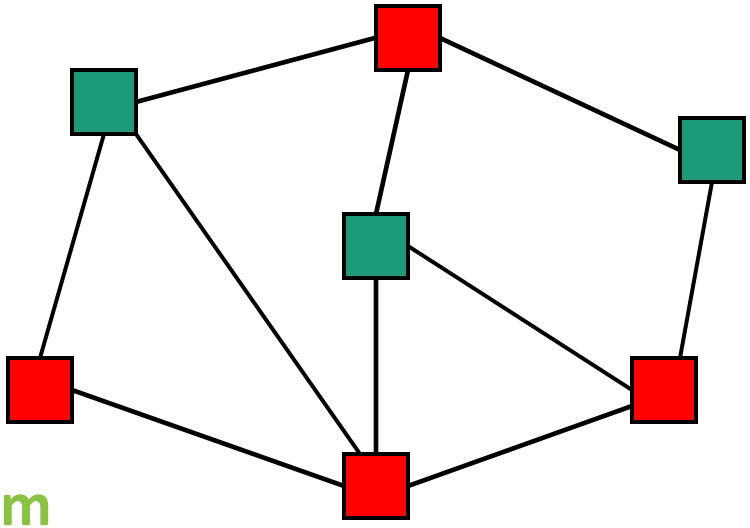
$O(2^{|V|})$ algorithm: Try every subset of vertices; pick smallest one

$O(|\mathbf{V}|^k)$ algorithm: Unknown, probably does not exist

Verifying a solution:

- Hmm, hard to verify an answer is *optimal* (smallest **|S|**)
- Can recast vertex cover as a *decision problem*

Vertex Cover: Decision Problem



Input: A graph (**V**,**E**) and a number **m**

Output: A subset **S** of **V** such that for every edge (**u**,**v**) in **E**, at least one of **u** or **v** is in **S** and **|S|=m** (if such an **S** exists)

$O(2^m)$ algorithm: Try every subset of vertices of size **m**

$O(m^k)$ algorithm: Unknown, probably does not exist

Verifying a solution: Easy, see if **S** has size **m** and covers edges

TSP

- The **travelling salesman problem** (**TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

Traveling Salesman

Input: A complete directed graph (**V**,**E**) and a number **m**

Output: A path that visits each vertex exactly once and has total cost < **m** if one exists

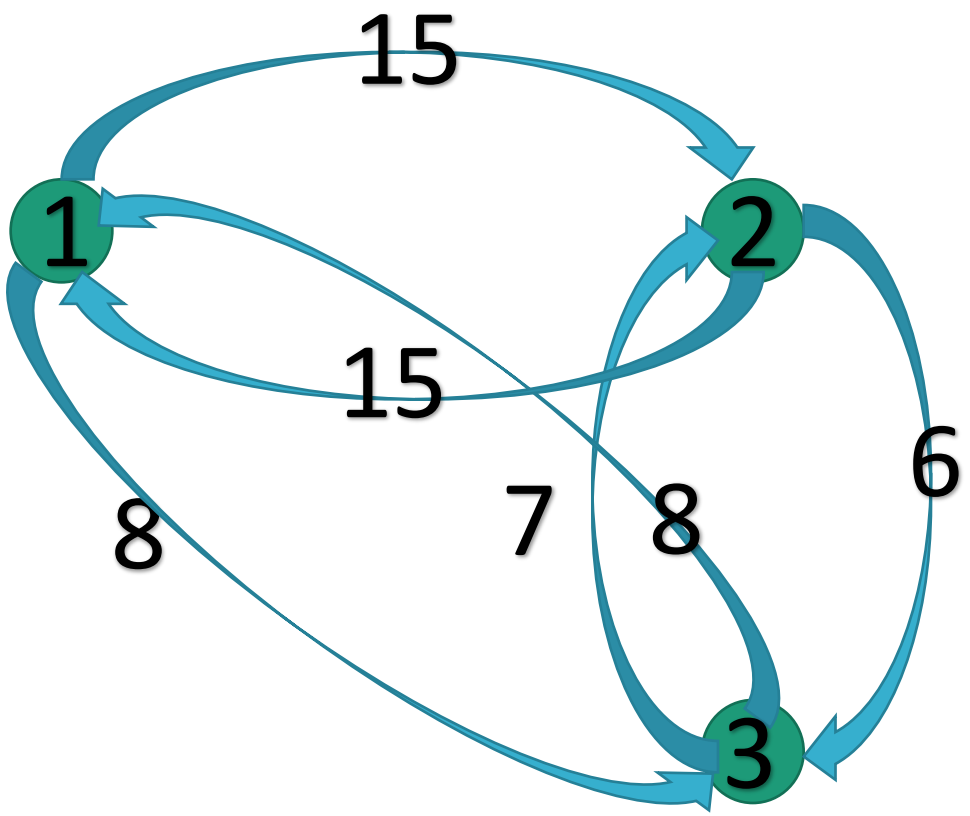
$O(2^{|V|})$ algorithm: Try every subset of vertices; pick smallest one

$O(|\mathbf{V}|^k)$ algorithm: Unknown, probably does not exist

Verifying a solution: Easy

TSP

$$C = \begin{pmatrix} 0 & 15 & 8 \\ 15 & 0 & 6 \\ 8 & 7 & 0 \end{pmatrix}$$



We are looking for a minimum path from 1 passing through all vertices and coming back to 1

Held–Karp algorithm

- Dynamic Programming Approach
- Basic idea: start from 1 and check all the possible ways to reach to neighbors and then check all the possible ways to go from neighbors to their neighbors. Find the minimum
- $O(n2^n)$ – Exponential time

Satisfiability

$$(\neg x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (x_2 \vee \neg x_4 \vee \neg x_5)$$

Input: a logic formula of size **m** containing **n** variables

Output: An assignment of Boolean values to the variables in the formula such that the formula is true

$O(\mathbf{m}*2^{\mathbf{n}})$ algorithm: Try every variable assignment

$O(\mathbf{m}^k\mathbf{n}^k)$ algorithm: Unknown, probably does not exist

Verifying a solution: Evaluate the formula under the assignment

The *Satisfiability* (SAT) Problem

- *Satisfiability* (SAT):

  Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?

$$((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

The *Satisfiability* (SAT) Problem

- *Satisfiability* (SAT):

  Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?

$$((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

- Seems simple enough, but no known deterministic polynomial time algorithm exists
- Easy to verify in polynomial time!

Double Implication(↔)

- **Definition 1.3.8**

The *biconditional* of p and q (p ↔ q)

- *p* if and only if *q*
- *p* is necessary and sufficient for q

- $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$
- **Truth Table for p ↔ q**

p	q	p → q	q → p	(p → q)∧(q → p)	p ↔ q
T	T	T	T	T	T
T	F	F	T	F	F
F	T	T	F	F	F
F	F	T	T	T	T

Example: CNF satisfiability

- *conjunctive normal form* (*CNF*) it is expressed as an AND of ***clauses***, each of which is the OR of one or more literals.

- Example:

$$(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee F) \wedge (F \vee \neg D)$$

A boolean formula is in ***3-CNF***, if each clause has exactly three distinct literals.

A disjunctive normal form (DNF) is a standardization (or normalization) of a logical formula which is a disjunction of conjunctive clauses; it can also be described as an OR of ANDs, a sum of products, or (in philosophical logic) a cluster concept.

Example: CNF satisfiability

- This problem is in *NP*.
- Nondeterministic algorithm:
  - Guess truth assignment
  - Check assignment to see if it satisfies CNF formula

$$(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee F) \wedge (F \vee \neg D)$$

- Truth assignments:

	A	B	C	D	E	F
1.	0	1	1	0	1	0
2.	1	0	0	0	0	1
3.	1	1	0	0	0	1
4.	... (how many more?)					

Checking phase: O(*n*)

More?

- Thousands of different problems that:
  - Have real applications
  - Nobody has polynomial algorithms for
- Widely believed: None of these problems have polynomial algorithms
  - For *optimal* solutions, but some can be *approximated*
- But: Nobody has ever proven that a single problem is:
  - In **NP**: A solution can be verified in polynomial time
  - And NOT in **P**: Cannot be solved in polynomial time

Review: P And NP Summary

- **P** : set of problems that can be solved in polynomial time

  Examples: Fractional Knapsack, …

- **NP** : set of problems for which a solution can be verified in polynomial time

  Examples: TSP,…, SAT

- Clearly **P** $\subseteq$ **NP**

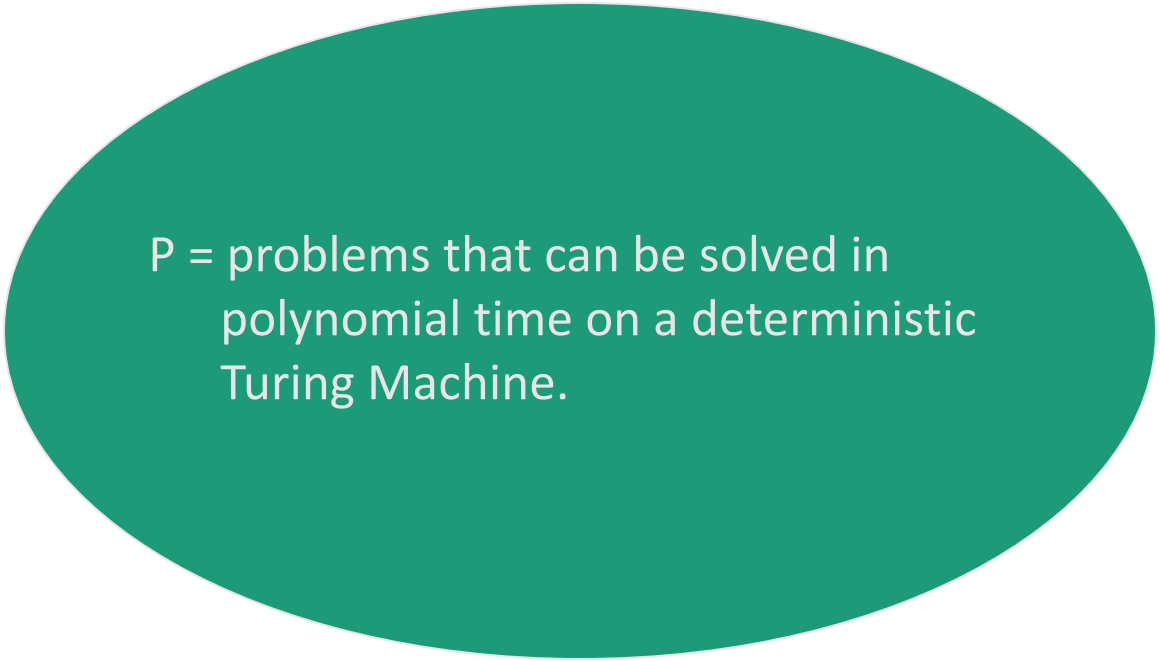
Review: P And NP Summary

- Open question for a long time: Does **P = NP**?
  - Most suspect not
  - An August 2010 claim of proof that P ≠ NP, by Vinay Deolalikar, researcher at HP Labs, Palo Alto, has flaws

Why it's called NP

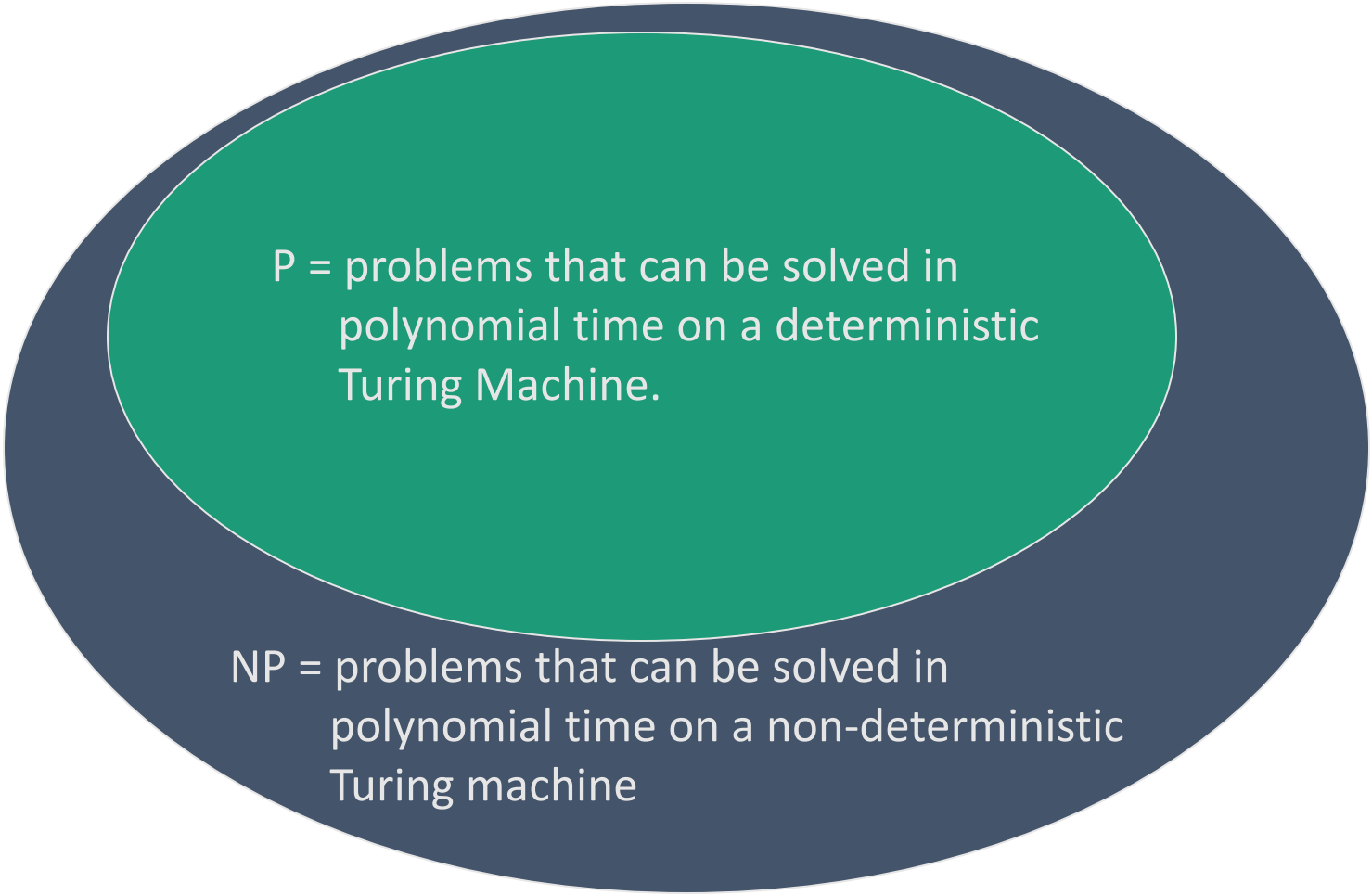
- There exists a polynomial time algorithm if the algorithm is allowed to make correct guesses at every step
  - This "guessing" is technically non-determinism
  - **NP** stands for non-deterministic polynomial time

P


P = problems that can be solved in polynomial time on a deterministic Turing Machine.

P, NP

P $\subseteq$ NP, because every problem in P has a solution in NP



Reduction

- A problem R can be *reduced* to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R
  - This rephrasing is called a *transformation*

- Intuitively: If R reduces in polynomial time to Q, R is “no harder to solve” than Q

NP-complete problems

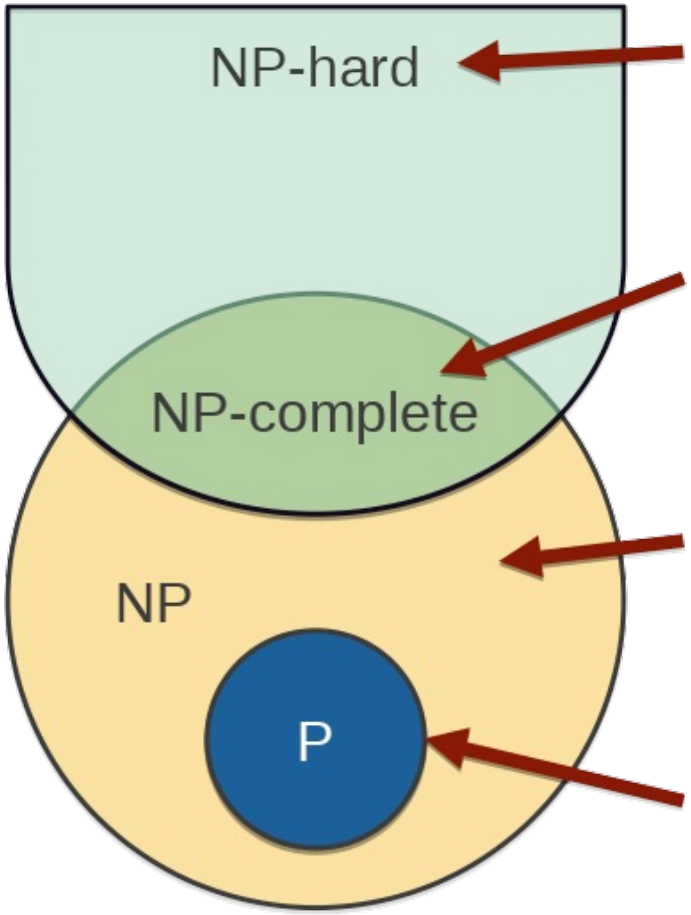
- A decision problem *D* is *NP*-complete if (if and only if)
  1. $D \in NP$
  2. every problem in *NP* is polynomial-time reducible to *D*

- If only (2) holds, then the problem is NP-hard
- Cook's theorem (1971): CNF-sat is *NP*-complete

NP-Hard and NP-Complete

- If R is *polynomial-time reducible* to Q, we denote this $R \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in$ **NP** are *polynomial-time* reducible to Q, then Q is *NP-Hard*
  - We say Q is *NP-Complete* if Q is NP-Hard and $Q \in$ **NP**
- If $R \leq_p Q$ and R is NP-Hard, Q is also NP-Hard

Venn Diagram of Common Complexity Classes



- Problems at least as hard as those in NP
  - Not necessarily decision problems
  - Ex.: Given weighted graph, find shortest-length Hamiltonian path?
- Hardest of the problems in NP
  - Ex.: Given set of integers, is there a subset whose sum is 0?
- "Hard" decision problems
  - Can be solved in polynomial time on a nondeterministic Turing machine
  - Solutions can be verified in polynomial time on a deterministic Turing machine
  - Ex.: Does given integer have prime factor whose last digit is 3?
- "Easy" decision problems
  - Can be solved in polynomial time on a deterministic Turing machine
  - Ex.: Does given matrix have an eigenvalue equal to 1.2?

What do we know about NP?

- Nobody knows if all problems in NP can be done in polynomial time, i.e. does P=NP?
  - one of the most important open questions in all of science.
  - Huge practical implications specially if answer is yes
- Every problem in P is in NP
  - one doesn't even need a certificate for problems in **P** so just ignore any hint you are given
- Every problem in NP is in exponential time
- Some problems in NP seem really hard
  - nobody knows how to prove that they are really hard to solve, i.e. $P \neq NP$

NP Completeness

We don't know how to prove any problem in NP is hard. So, let's find hardest problems in NP.

NP-hard: A problem B is NP-hard if for any problem $A \in NP$, we have $A \leq_p B$

NP-Completeness: A problem B is NP-complete if B is NP-hard and $B \in NP$.

Motivations:

- If $P \neq NP$, then every NP-Complete problems is not in P. So, we shouldn't try to design Polytime algorithms
- To show $P = NP$, it is enough to design a polynomial time algorithm for just one NP-complete problem.

Relative Complexity of Problems

- Want a notion that allows us to compare the complexity of problems

- Want to be able to make statements of the form

  "If we could solve problem **B** in polynomial time then we can solve problem **A** in polynomial time"

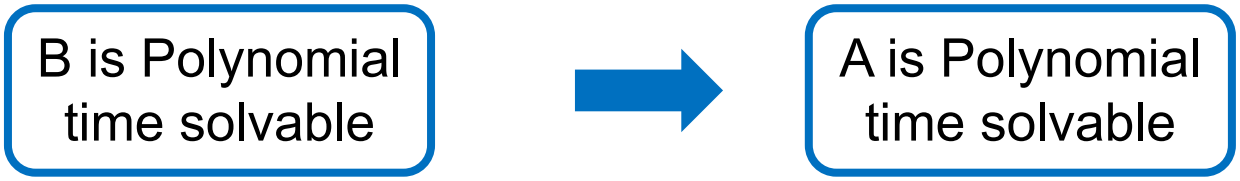
  "Problem **B** is at least as hard as problem **A**"

Polynomial Time Reduction

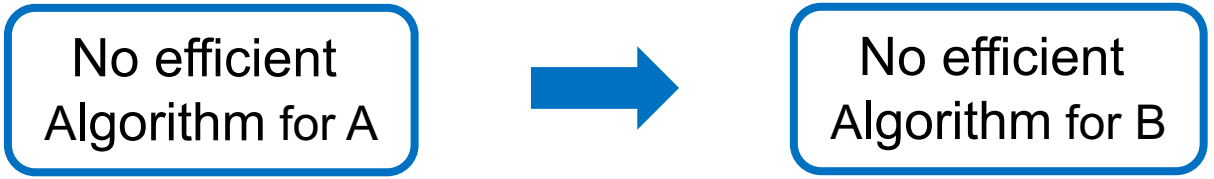
Def $A \leq_P B$: if there is an algorithm for problem A using a 'black box' (subroutine) that solves problem B s.t.,

- Algorithm uses only a polynomial number of steps
- Makes only a polynomial number of calls to a subroutine for **B**

So,



Conversely,



In words, B is as hard as A (it can be even harder)

Reductions

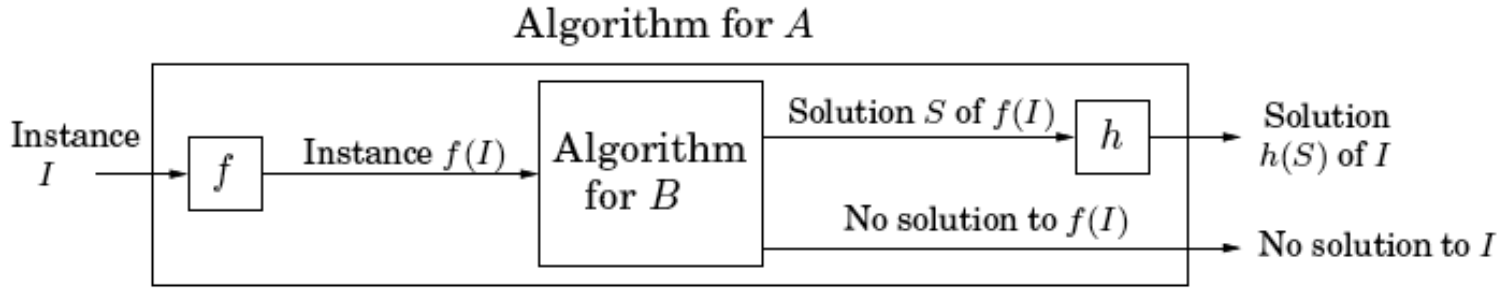
Here, we often use a restricted form of polynomial-time reduction often called Karp reduction.

$A \leq_p B$: if and only if there is an algorithm for A given a black box solving B that on input x

- Runs for polynomial time computing an input f(x) of B
- Makes one call to the black box for B for input f(x)
- Returns the answer that the black box gave

We say that the function f(.) is the reduction

Reductions Again



- As long as *f* and *h* are polynomial time translations (have complexity ≤ *B*) then algorithm *A* will have the same complexity as algorithm *B*
- We say *A* reduces to *B*
- A problem is *NP-Complete* if all other problems in NP reduce to it

Examples of Problems in NP vs P

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

- For the problems in P we can find diverse algorithmic approaches to solve them efficiently with different complexities in P.

- There is a consistency amongst the problems in NP which makes them all equally hard to solve

Unsolvable Problems

- There are problems in higher complexity classes than NP
  - Guaranteed exponential, super-exponential, etc.
  - List all possible paths in a TSP problem
  - List the power-set of all possible paths in TSP, etc.
- There is an even larger infinite class of unsolvable problems regardless of time available
  - For an arbitrary polynomial equation in many variables e.g.:
  - $x^2yz^3 + 3y^2z - 4xy^5z^2 = 7$
  - Are there integer values of the variables which solve it? (x, y, and z in this case)
- Problems as Functions
  - Regularity vs Random
  - Also many regular problems that are not solvable: halting, etc.

List of NP-complete problems

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

https://en.wikipedia.org/wiki/Karp%27s_21_NP-complete_problems