CS 231: Algorithmic Problem Solving

Naomi Nishimura

Module 6

Date of this version: February 10, 2020

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Upper and lower bounds on problems

The **complexity** of a problem is a way of describing the worst-case cost of the best algorithm for the problem.

An **upper bound** of $O(f(n))$ on a problem means that there exists an algorithm correctly solving the problem that in the worst case runs in time $O(f(n))$.

An upper bound on the worst-case running time of an algorithm that solves a problem is also an upper bound on the problem.

A **lower bound** of $\Omega(g(n))$ on a problem means that **any** algorithm that correctly solves the problem must use $\Omega(g(n))$ time in the worst case.

CAUTION!

A lower bound on the worst-case running time of an algorithm may not be a lower bound on the problem, as it only refers to **one** algorithm. There might exist another algorithm that runs in less time.

Determining when algorithms are the best

Sample algorithms:

- Searching in an ordered list in $O(\log n)$ worst-case time using binary search
- Finding the minimum in a list in $O(n)$ worst-case time
- Sorting in $O(n \log n)$ worst-case time using mergesort
- Determining a minimum spanning tree in $O(m \log n)$ time
- Solving TSP in $O(n^n)$ time

Can we do better?

What is "good enough"?

Aiming for polynomial time

A **polynomial in** n is a function of the form $\sum_{i=0}^{k} c_i n^i$ for constants c_i and some positive integer k.

Similarly, we can talk about a polynomial on more than one variable, where terms are of the form $c_{i,j} n^i m^j$ for two variables.

Note: k cannot be a function of n. The TSP algorithm mentioned in our examples was not a polynomial-time algorithm.

Reasons polynomial is good:

- Computable in our lifetimes (maybe). (Most polynomial-time algorithms are cubic, quadratic, or even linear.)
- Robust.

Robustness

Robustness means that a problem doesn't get booted out of the class due to a small change in the definition.

Example: A different graph implementation should not result in the complexity of problems changing.

Reasons behind robustness:

- The sum of two polynomials in n is a polynomial in n.
- The product of two polynomials in n is a polynomial in n.
- If $f(n)$ and $g(n)$ are both polynomials in n, then so if $f(g(n))$.

$f(n) = n^3 + 2n + 5$, $g(n) = 2n^2$,
$f(g(n)) = (2n^2)^3 + 2(2n^2) + 5 = 8n^6 + 4n^2 + 5$

A polynomial output can be used as an input for another problem.

Complexity classes

A **complexity class** is a set of problems that can be solved with specified bounds on resources using specific models of computation.

A (decision) problem is in the complexity class **P** if can be solved using an algorithm with worst-case running time that is polynomial in the size of the input.

A problem is considered to be **tractable** if it is known to be in P and **intractable** otherwise.

Other interesting running times:

- **subexponential** - better than just trying all options
- **linear** - proportional to size of input, which needs to be read
- **sublinear** - not all items have to be read

Searching for better algorithms

Goals:

- Determine complexity of problems by finding matching lower bounds
- Attempt to find better algorithms when matching lower bounds have not been found
- Recognize when polynomial-time algorithms are likely to be elusive

Finding a lower bound

A function $g(n)$ is a **lower bound** on a function $f(n)$ if $g(n)$ is no greater than $f(n)$.
Observations:

- The larger the lower bound, the most information it gives. (A constant time lower bound is not very useful.)
- If we already know what $f(n)$ is, we already have a precise lower bound.

Key operations

A **key operation** is a step that is representative of the computation overall.

Properties of key operations:

- Can be constant-time operations
- Represent or dominate other operations
- Number of key operations should give a function of the input size

Examples of key operations:

- Access one data item (e.g. grid entry)
- Compare two data items (outcome $=$ or $\neq$)
- Compare two two items (outcome $=$, $<$, or $>$)
- Determine if two vertices in a graph are adjacent

A model of computation might be designed around key operations; this is an example of a strong model, good for lower bounds, as discussed in Module 2.

Types of lower bounds

- **Information theory lower bounds**
  - Key operations: Comparisons
  - Counting argument
- **Adversary lower bounds**
  - Key operations: Accessing data items, other operations
  - Computational game
- **Reductions**
  - Relates difficulty of two problems
  - Relies on existing lower bound on one of the problems

Some of the same bounds can be reinterpreted in different ways.

Decision trees

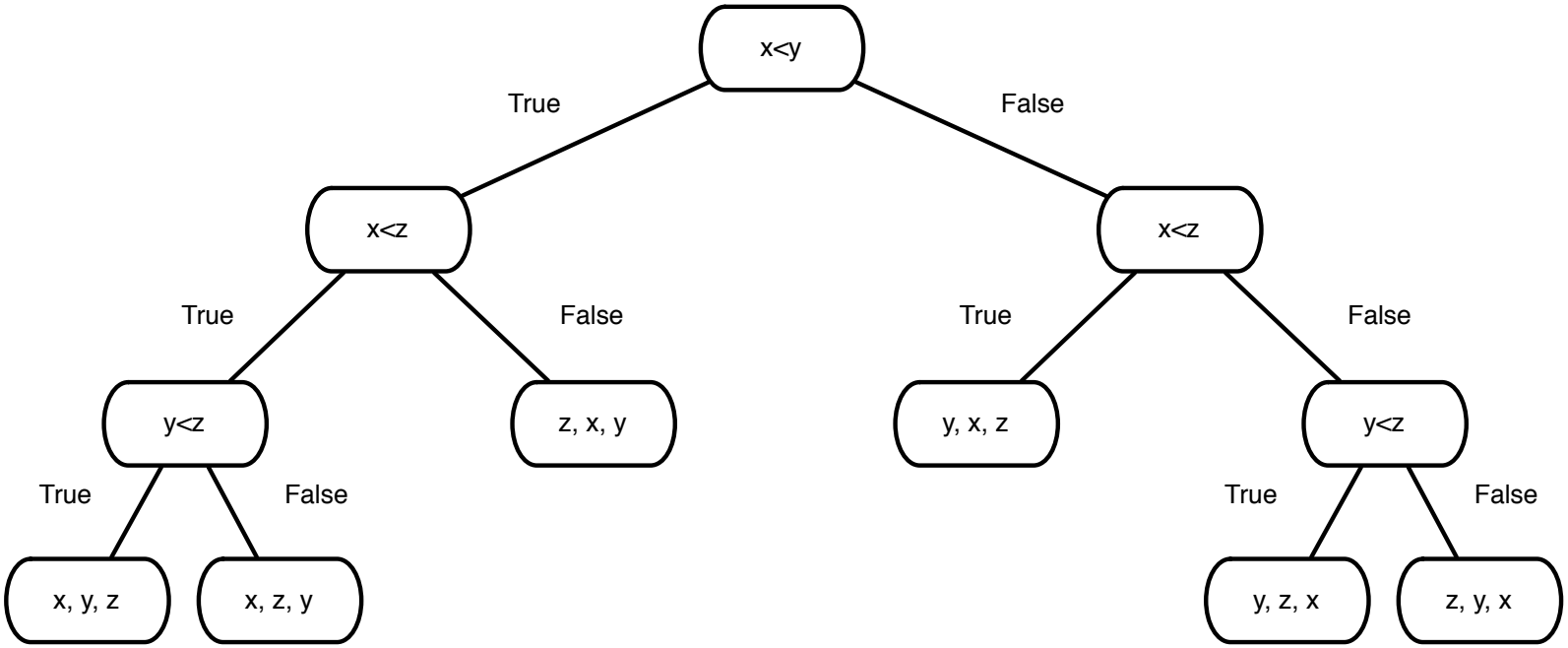
A **decision tree** is a model of computation that demonstrates the use of comparisons as key operations in a **comparison-based algorithm** (one in which operations on inputs are limited to comparing them to each other).

Tree terminology:

- A **tree** is made up **nodes** connected by **edges**.
- Each edge connects a **parent** to a **child**.
- Nodes with the same parent are **siblings**.
- The node without a parent is the **root**.
- A node without children is a **leaf**. A node that is not a leaf is an **internal node**.

Each internal node in the tree represents a comparison and each leaf represents an output. The children of a node represent the possible next course of action depending on the outcome of the comparison.

Example of a decision tree for SORTING



Using a decision tree to analyze an algorithm

For the decision tree representing a particular algorithm, the number of internal nodes on the path from the root to a leaf gives the number of comparisons made to reach that leaf.

The number of comparisons is a lower bound on the total running time for that output.

A lower bound on the worst-case running time for the algorithm is found by looking at the longest path from the root to a leaf, or the height of the tree.

Using a decision tree to analyze a problem

If any algorithm for a problem can be represented using a decision tree, then we can find a lower bound on the complexity of the problem (for comparison-based solutions) by finding a lower bound on the height of **any** decision tree for the problem.

(Or the bound might be restricted to comparison-based algorithms for the problem.)

If a decision tree must have height at least h, then any algorithm must take time at least $\Omega(h)$ in the worst case.

Decision tree lower bounds

A decision tree is a binary tree.

A binary tree with ℓ leaves must have height $\Omega(\log \ell)$.

Searching n values can have n possible outputs, hence $\Omega(\log n)$ is a lower bound.

Sorting n values can have $n!$ possible outputs, hence $\Omega(n \log n)$ is a lower bound.

Explanations of logarithms can be found in the math guide.

Note: These lower bounds only apply to comparison-based algorithms. They do not apply if we can compute various functions on the values of inputs.

Questions and answers by the algorithms and the adversary

For a specified key operation or sets of key operations, the outcomes of the operations can be viewed as "answers" given by an adversary to "questions" posed by the algorithm.

View computation as a game between the algorithm and an adversary, according to the following rules:

- The algorithm asks questions to try to figure out which input (and hence which output) it is.
- The adversary answers questions such that there is at least one input consistent with the answers.

Strategies:

- Algorithm: Ask questions to reduce the set of possibilities as much as possible
- Adversary: Answer questions to keep the set of possibilities as big as possible

Adversary lower bounds

An **adversary strategy** is an algorithm (with no limits on computation time or other resources) that the adversary uses to determine what answer to give to any question posed by the algorithm. However, it cannot see into the future to know what the algorithm will do next.

When the algorithm has chosen an output, the adversary can choose any input consistent with the answers it gave to the questions.

The algorithm wins if the output is correct for the chosen input, and loses otherwise.

Proving an adversary lower bound

Recipe for determining an adversary lower bound:

1. Specify an adversary strategy.
2. Determine a number of steps T that any correct algorithm must take.
3. Show that after $T-1$ steps of any algorithm, there will be at least two inputs consistent with the answers given by the adversary, and that they yield different outputs.

Note: The bound only applies to algorithms that are restricted to accessing the input using the specified key operation(s).

Using a proof by contradiction in an adversary lower bound

- Suppose that the statement "Any correct algorithm must use at least T steps." is false.
- There must be a correct algorithm that uses at most $T-1$ steps.
- Show that no matter what the algorithm learns in $T-1$ steps, there will still remain at least two possible inputs consistent with the answers given by the adversary to the key operations, each yielding a different output.
- Whichever output the algorithm chooses, the adversary can select an input that yields a different output.
- The fact that the algorithm has chosen the wrong output contradicts the assumption that the algorithm is correct.
- We can then conclude that our assumption that the statement was false was wrong.
- Consequently, we can conclude that any correct algorithm must use at least T steps.

Adversary strategy for One odd

Any odd

Input: A nonempty sequence of numbers

Output: Yes or no, answering "Does the sequence contain an odd number?"

Algorithm step: Examine a number

Step 1: Specify an adversary strategy.

- Keep a list of items examined so far.
- Each time an item is examined, make it be even.

Adversary lower bound for ONE ODD

Step 2: Determine a number of steps T that any correct algorithm must take.

Claim: Any correct algorithm must take at least n steps.

Step 3: Show that after $T-1$ steps of any algorithm, there will be at least two inputs consistent with the answers given by the adversary, and that they yield different outputs.

Proof:
Suppose the algorithm takes only $n-1$ steps. Then at least one item has not been revealed.

If the algorithm answers "yes", the adversary makes all items be even so that the algorithm is incorrect.

If the algorithm answers "no", the adversary makes an unrevealed item be odd so that the algorithm is incorrect.

Adversary strategy for SORTING

Algorithm step: Ask if one number is larger than another number

Step 1: Specify an adversary strategy.

- Keep a list of all the possible permutations that are consistent with the answers so far.
- Each time a comparison is made, split the lists of permutations into those consistent with "yes" and those consistent with "no".
- Choose the answer (yes or no) that results in the larger list, throwing out the other list.

Adversary lower bound for SORTING

Step 2: Determine a number of steps T that any correct algorithm must take.

Claim: Any correct algorithm must ask $\lfloor \log n! \rfloor$ questions.

Step 3: Show that after $T-1$ steps of any algorithm, there will be at least two inputs consistent with the answers given by the adversary, and that they yield different outputs.

Proof:

Suppose the algorithm asks fewer questions.

Each time a question is asked, at most half of the permutations can be discarded.

After fewer than $\lfloor \log n! \rfloor$ questions, there will be more than one permutation left.

No matter which permutation the algorithm produces, the adversary can select one that will make it be wrong.

Note: We can use the same argument to convert any information theory lower bound to an adversary lower bound.

Adversary strategy for MAXIMUM

Algorithm step: Ask if one number is larger than another number

Step 1: Specify an adversary strategy.

- Keep track of the values of numbers that have been revealed so far.
- For each comparison, choose and reveal a value for any number without a revealed value such that the values are consistent with the values revealed so far.

Adversary lower bound for MAXIMUM

Step 2: Determine a number of steps T that any correct algorithm must take.

Claim: Any correct algorithm must make at least $\lceil n/2 \rceil$ comparisons.

Step 3: Show that after $T-1$ steps of any algorithm, there will be at least two inputs consistent with the answers given by the adversary, and that they yield different outputs.

Proof:

Suppose the algorithm makes fewer comparisons.

Each time a question is asked, at most two numbers are given values.

After fewer than $\lceil n/2 \rceil$ comparisons, at least one number x hasn't been given a value.

If the algorithm chooses x as the maximum, the adversary gives x a value that is not the maximum.

If the algorithm does not choose x as the maximum, the adversary gives x the maximum value.

Better adversary strategy for MAXIMUM

- Keep track of which numbers have "lost" a comparison and their relative order.
- If nothing is known, a number is a "nonloser".
- For each comparison between
  - a nonloser and a loser, make the loser lose
  - two losers, answer consistently with the order
  - two nonlosers, answer arbitrarily

Better adversary lower bound for MAXIMUM

Claim: Any correct algorithm must make $n-1$ comparisons.

Proof:

Suppose the algorithm makes fewer comparisons.

Each time a comparison is made, at most one new number is marked as a loser.

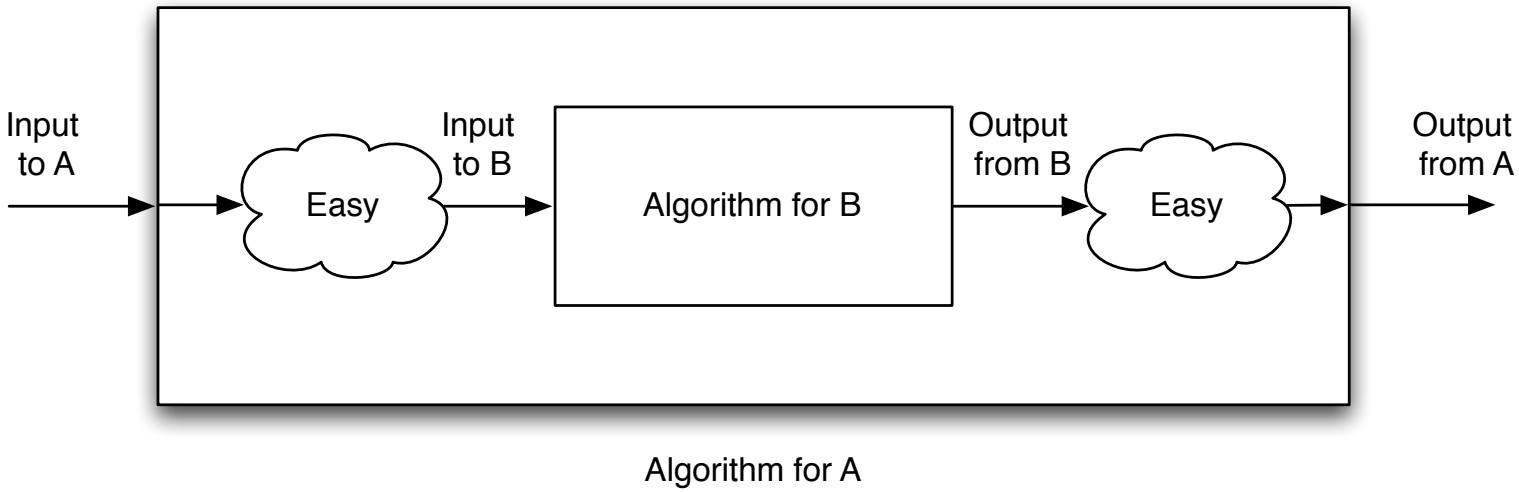
After fewer than $n-1$ questions, there will at least two nonlosers left.

No matter which nonloser the algorithm produces, the adversary can select the other one to make the algorithm be wrong.

Relating problems

Problem A can be **reduced** to problem B if we can solve A using a procedure for B and at most polynomial extra time.

Note: We saw this idea in the general problem-solving technique "Use what you already know."



"A is no harder than B"

- If A can be reduced to B and B is "easy", then A is "easy" too.
- If A can be reduced to B and A is "hard", then B must be "hard" too.

Details of the statements

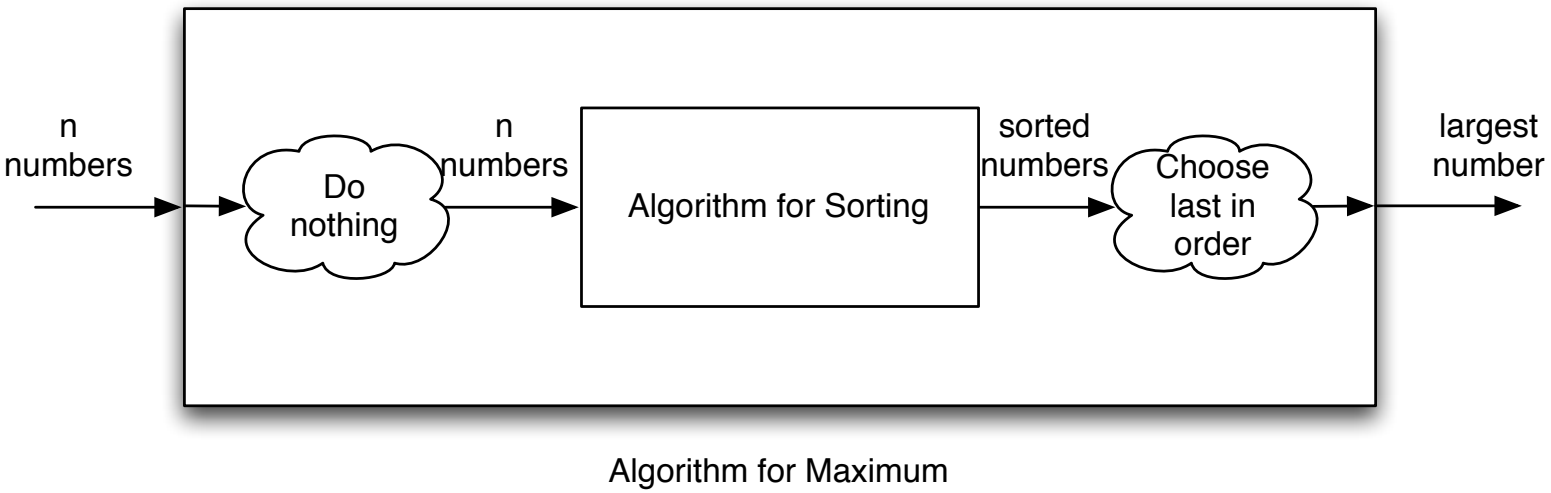
- If A can be reduced to B and B is "easy", then A is "easy" too.
- If A can be reduced to B and A is "hard", then B must be "hard" too.

The first statement results from all parts of the algorithm being "easy".

We can use a proof by contradiction for the second statement.

- Suppose that the statement is false.
- Then the following all are true:
  1. A can be reduced to B
  2. A is "hard"
  3. B is "easy"
- But by the first statement, parts (1) and (3) imply that A is "easy".
- A is "easy" contradicts (2)
- Thus, our supposition was incorrect, and the statement must be true.

A reduction from MAXIMUM to sorting



"Maximum is no harder than Sorting"

Using reductions to relate problems

Path constructive optimization

Input: A graph G and vertices u and v

Output: A path of minimum length from u to v

Path Decision

Input: A graph G and vertices u and v, and an integer k

Output: Yes or no, answering "Is there a path from u to v of length at most k?"

Definition

Two problems are **equivalent** in difficulty if each can be reduced to the other.

Equivalence of path problems

Reducing decision to constructive optimization:

- Use the inputs G, u, and v from decision as the input to constructive optimization
- Determine the length of the output
- Return "Yes" if it is at most k, "No" otherwise

Reducing constructive optimization to decision:

- Use the decision algorithm repeatedly using binary (or linear) search to determine the length ℓ of the shortest path
- Find a shortest path by using the decision algorithm on G, w, v, and $\ell - 1$ for each neighbour w of u, selecting a w for which the answer is "Yes", and then continuing the search from w

P is defined in terms of decision problems; we focus on those now.

NP and verification

Definition

A **verification algorithm** for a decision problem has two inputs:

- an instance of a problem
- a certificate

A problem can be **verified in polynomial time** if for each yes-instance there exists a polynomial-size **certificate** which can be used to obtain the output "Yes".

The verification algorithm needs to check both that what is provided is a feasible solution of the optimization version of the problem and that it satisfies the conditions stated.

Note: This is only a statement about yes-instances. There is no guarantee for no-instances.

The complexity class NP

Definition

NP is class of decision problems that can be verified in polynomial time using a polynomial-size certificate.

Aside

NP stands for **nondeterministic polynomial**; one can think of the certificate as being formed of nondeterministic "guesses".

Recipe for membership in NP:

1. Give a polynomial-size certificate for each yes-instance.
2. Give a polynomial-time verification algorithm.
3. Show that the algorithm answers "Yes" for any yes-instance and its certificate.
4. Show that the algorithm is not fooled by false certificates for any no-instances.

3-Colouring is in NP

3-Colouring

Input: A graph G

Output: Yes or no, answering "Is G 3-colourable?"

1. Give a polynomial-size certificate for each yes-instance. *Sequence of colours*
2. Give a polynomial-time verification algorithm. *Check length; check each edge to make sure its endpoints have different colours; make sure the total number of colours is at most three.*
3. Show that the algorithm answers "Yes" for any yes-instance and its certificate. *The certificate for a yes-instance will have edges with different coloured endpoints and a total of at most three colours.*
4. Show that the algorithm is not fooled by false certificates for any no-instances. *Any colouring of a no-instance will either use at least four colours or will give the same colour to both endpoints of an edge.*

Composite is in NP

Composite

Input: A positive integer N

Output: Yes or no, answering "Is N composite?"

1. Give a polynomial-size certificate for each yes-instance. *Two integers K and L, each greater than* 1
2. Give a polynomial-time verification algorithm. *Check if N is the product of K and L.*
3. Show that the algorithm answers "Yes" for any yes-instance and its certificate. *The certificate for a yes-instance will yield N as the product.*
4. Show that the algorithm is not fooled by false certificates for any no-instances. *There is no pair of integers, each greater than* 1, *whose product will equal a prime number.*

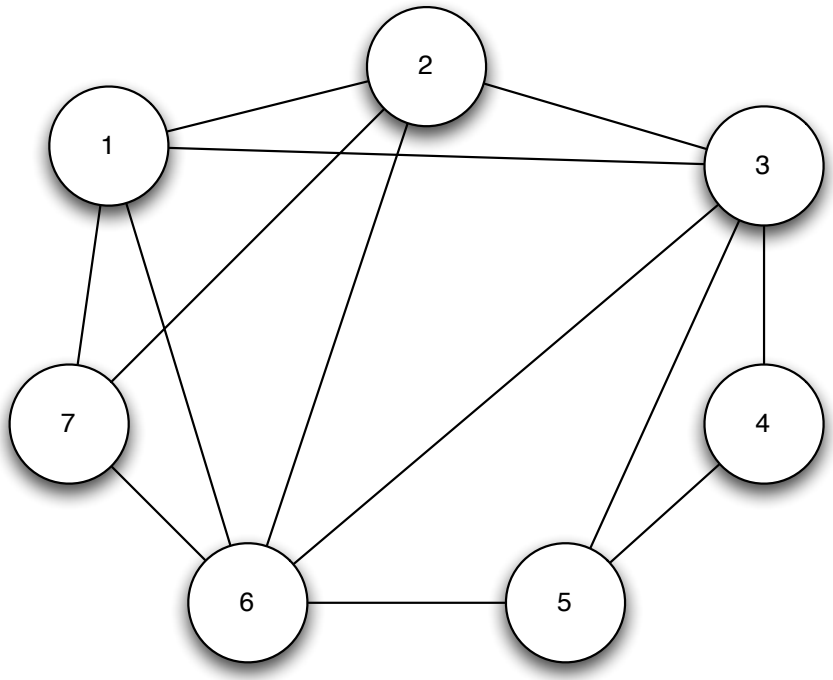
Hamiltonian Cycle

A **Hamiltonian cycle** in a graph is an ordering of all the vertices that forms a cycle.

Hamiltonian Cycle

Input: A graph G

Output: Yes or no, answering "Does G have a Hamiltonian cycle?"



Hamiltonian Cycle is in NP

1. Give a polynomial-size certificate for each yes-instance. *A sequence of vertices*
2. Give a polynomial-time verification algorithm. *Check length, that each vertex appears exactly once in the sequence, and that there is an edge between each pair of vertices in the cyclic ordering.*
3. Show that the algorithm answers "Yes" for any yes-instance and its certificate. *The certificate for a yes-instance will include edges between each pair of vertices.*
4. Show that the algorithm is not fooled by false certificates for any no-instances. *Any ordering of vertices of a no-instance will lack at least one edge or at least one vertex.*

P versus NP

Theorem

Every problem that is in P is also in NP.

Our "verification algorithm" can ignore the certificate and still solve the problem.

Major open question

Is P=NP?

Why does anyone care?

If so, we can solve all these problems in polynomial time.
If not, we know all these problems are "hard" to solve.

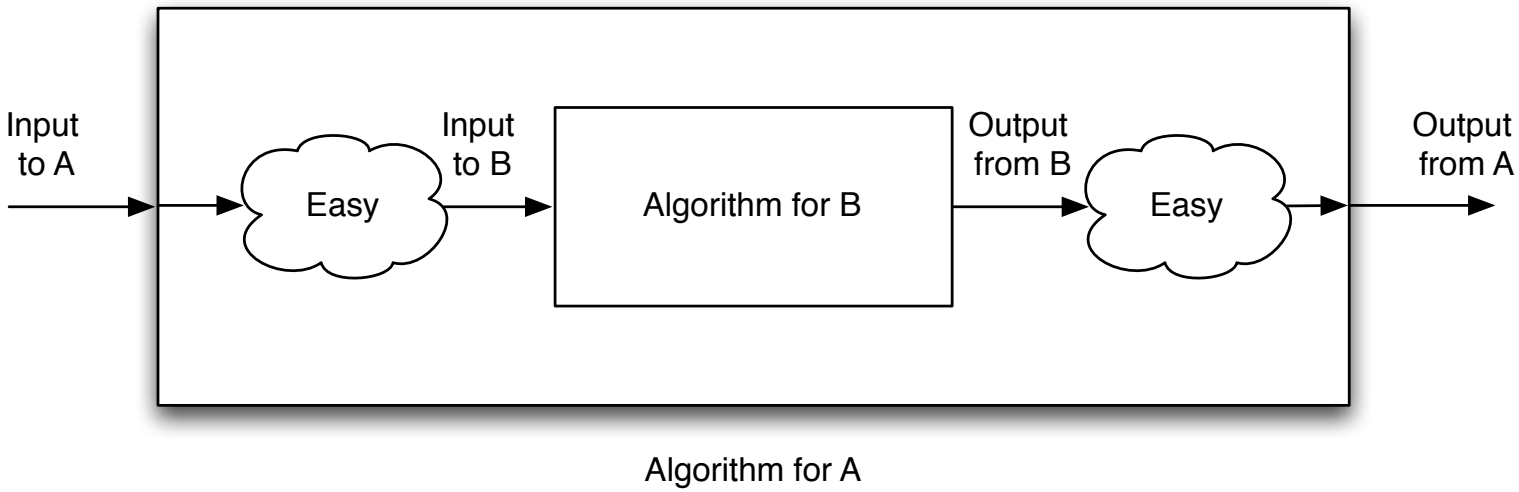
The "hardest" problems in NP

Definition

B is **NP-hard** if for every $A \in NP$, A is reducible to B.

Definition

L is **NP-complete** if L is in NP and L is NP-hard.



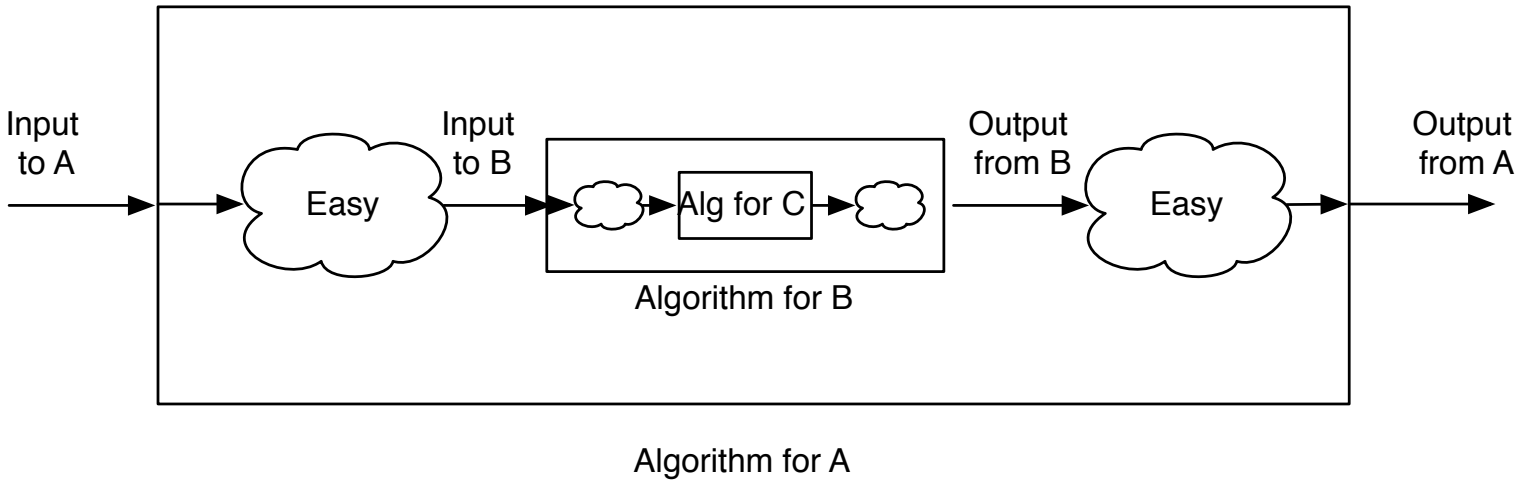
- If any NP-complete problem is in P, then P = NP.
- If any NP-complete problem is not in P, then P $\neq$ NP.

Showing a problem C is NP-hard

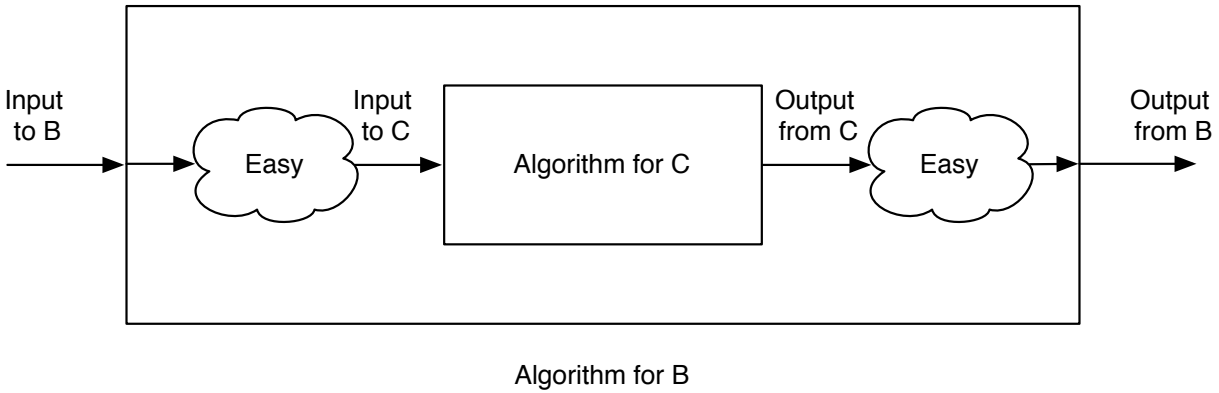
Hard way: Take every A in NP and show a reduction from A to C.

Easy way:

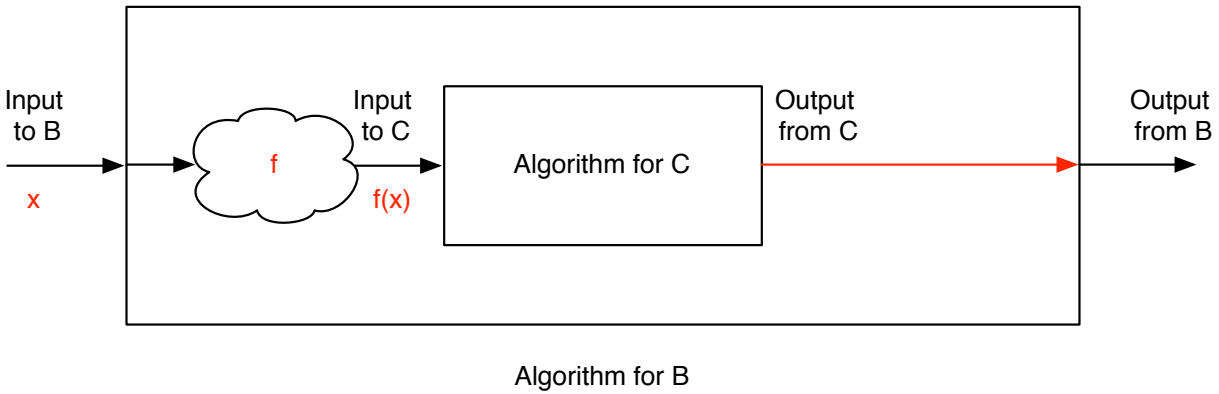
- Take an NP-hard problem B.
- Show B reduces to C.
- Then every A in NP reduces to B reduces to C.



A simplified type of reduction



- Transform an input x to B into an input $f(x)$ of C using a function f.
- Ensure that the output on $f(x)$ (using C) is the same as the output on x (using B).



Using the simple reduction

Equivalent statements (using contrapositives):

- x is a yes-instance for B if and only if $f(x)$ is a yes-instance for C
- If x is a yes-instance for B, then $f(x)$ is a yes-instance for C AND if $f(x)$ is a yes-instance for C, then x is a yes-instance for B.
- If x is a yes-instance for B, then $f(x)$ is a yes-instance for C AND if x is NOT a yes-instance for B, then $f(x)$ is NOT a yes-instance for C.
- If x is a yes-instance for B, then $f(x)$ is a yes-instance for C AND if x is a no-instance for B, then $f(x)$ is a no-instance for C.

Problem: we need a "first" NP-complete problem B.

Aside

- Cook's Theorem established the first problem.
- Karp proved that many common problems are NP-complete.

NP-completeness recipe

Recipe for NP-completeness:

1. Prove C is in NP.
2. Select B that is known to be NP-complete.
3. Give an algorithm to compute a function f mapping each instance of B to an instance of C (it needn't map to all of C).
4. Prove that for any string x, if x is a yes-instance for B then f(x) is a yes-instance for C.
5. Prove that for any string x, if f(x) is a yes-instance for C then x is a yes-instance for B.
6. Prove that the algorithm computing f runs in polynomial time.

Caveats:

- Make sure the reduction is in the right direction.
- Be sure to ensure that f maps *all* instances of B.
- Don't forget Step 6.
- Don't believe Step 3 without checking Steps 4 and 5.

Case study: Monitoring all connections

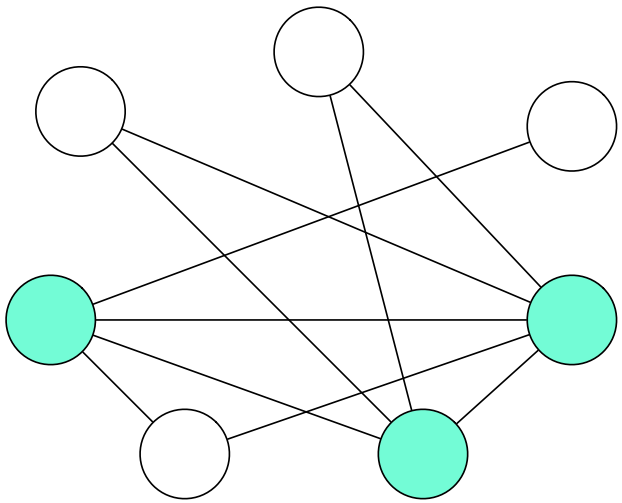
Determine the smallest number of "guards" needed to monitor all possible connections.

A **vertex cover** of size k in a graph G is a subset V' of the vertices such that at least one endpoint of each edge is in V'.

Vertex Cover

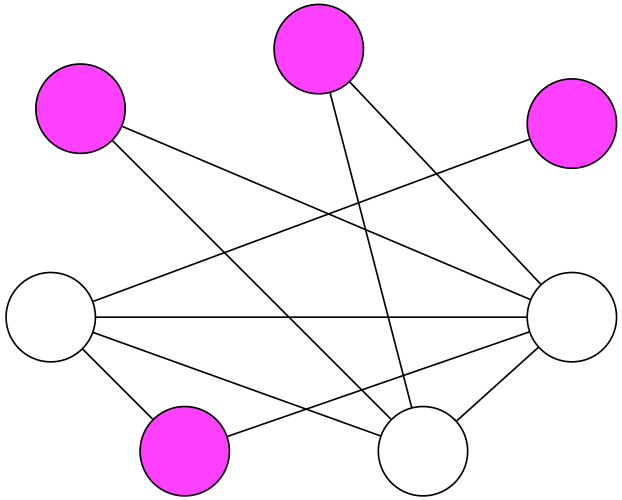
Input: A graph G and an integer k

Output: Yes or no, answering "Does G have a vertex cover of size at most k?"



Independent Set

An **independent set** in a graph is a subset V' of vertices such that no two vertices in V' are connected by an edge.



You can assume that the following problem is NP-complete:

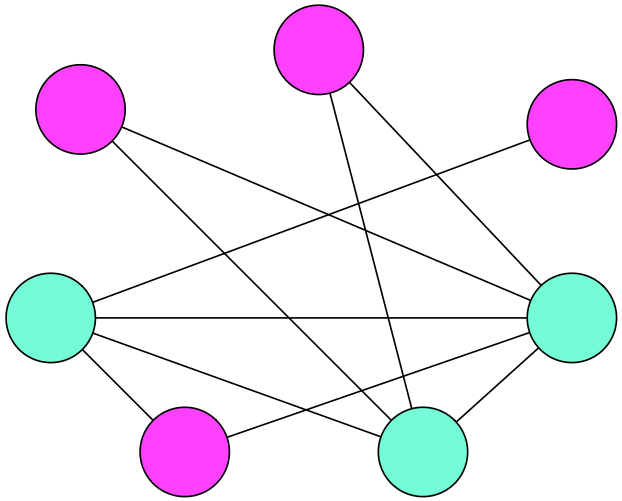
Independent Set

Input: A graph G and an integer k

Output: Yes or no, answering "Does G have an independent set of size at least k?"

Relating independent set and vertex cover

If there is an independent set in G of size at least k, then there is a vertex cover in G of size at most $n - k$.

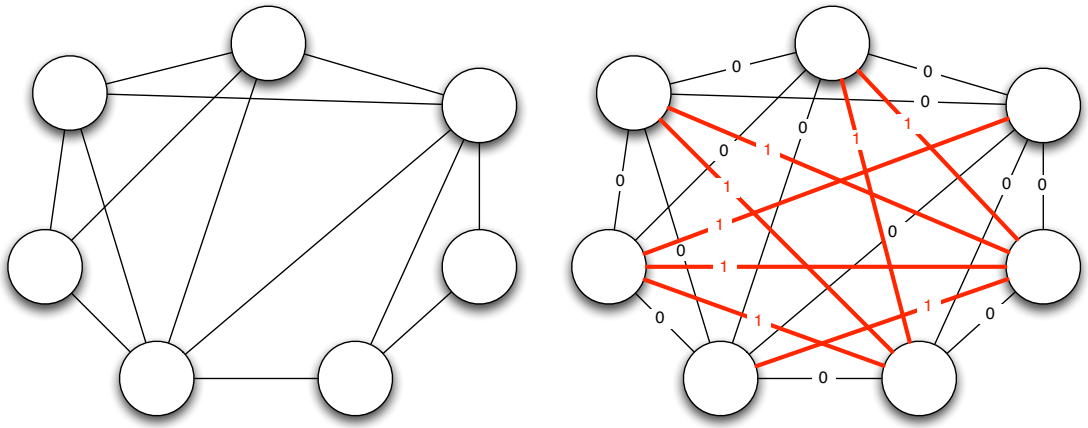


TSP is NP-complete

You can assume Hamiltonian Cycle is NP-complete

Step 3: Give an algorithm to compute a function f mapping each instance of B to an instance of C (it needn't map to all of C).

Given input G for Hamiltonian Cycle, create the instance of TSP with a weighted graph G' and $k = 0$, where G' has weight 0 for each edge in G and weight 1 for each edge not in G.



Encodings revisited

A problem can be solved in **pseudopolynomial time** if the running time is polynomial in the value of the input but exponential in the length of encoding of the input.

Our dynamic programming solution for knapsack is an example, as the size depended on the value of the bound.

A problem is **strongly NP-complete** if all numerical values are bounded by a polynomial in the length of the input.

Module summary

Topics covered:

- Polynomial time
- The complexity class P
- Lower bounds
- Key operations
- Decision trees
- Information theory lower bounds
- Adversary lower bounds
- Reductions
- Verification
- The complexity class NP
- NP-completeness
- Pseudopolynomial time and strong NP-completeness